DØOM User Guide

H. Greenlee, J. Hobbs, S. Snyder, V. White

April 30, 2001

For `v00-27-01`

Contents

1 Overview of DØOM

DØOM is an object persistency system for C++ classes. DØOM sits on top of an underlying I/O package, such as the DSPACK [1] (for sequential files) or a database. DØOM consists of the following elements.

1. Object model classes.
2. Preprocessor.
3. User I/O classes.
4. Dictionary classes.
5. DSPACK interface classes.

Normally, users interact directly only with the first three of the above five items. The last two items are primarily intended for internal use by DØOM. In particular, package specific I/O calls are hidden from the user. The underlying I/O mechanism can be changed without affecting the DØOM user interface.

In addition to this guide, you should also consult the C++ headers for the classes which you are going to use. In most cases, the details of the interface are described only in the headers.

1.1 The DØOM Object Model

The DØOM object model is loosely based on the ODMG object model for object oriented databases [2]. The DØOM object model requires that persistent classes have the following properties.

1. Persistent classes inherit, directly or indirectly, from the persistent base class `d0_Object`.
2. Persistent classes are composed of the following elements:
   (a) Atomic types (`int`, `float`, `d0_Int`, `d0_Float`, etc.).
   (b) C++ fixed length arrays.

(c) DØOM container classes (e.g. `d0_Vector<T>`).

(d) The C++ standard container classes `deque`, `list`, `vector`, `set`, `multiset`, `map`, `multimap`, `stack`, `queue`, `priority_queue`, and `valarray`.

(e) The additional standard C++ types `bitset` and `complex`.

(f) DØOM strings (`d0_String`) or the C++ standard string class `string` (which can also be called `basic_string`).

(g) DØOM reference classes (smart pointers, `d0_Ref<T>`).

(h) C++ bare pointers and `auto_ptr`.

(i) Literal classes (non-persistent user-defined classes contained within persistent classes). Literal classes do not need to be derived from `d0_Object`, but must be composed of the types allowed in persistent classes.

Certain restrictions apply to the types that can be used with the standard DØOM template classes (containers and references). Refer to sections 2.2 and 2.4 for details.

Note that data member names starting with two underscores are reserved.

1.2 The DØOM Preprocessor and Dictionary Classes

The DØOM preprocessor is borrowed from the cint C++ interpreter, which is distributed as part of the ROOT system [3]. The dictionary classes contain information about the structure of user classes, or in other words they contain class metadata. The DØOM preprocessor, `d0cint`, analyzes user's header files and generates the code necessary to create the dictionary classes. Users do not normally invoke the dictionary classes directly. Rather, the DØOM I/O classes use the information in the dictionary classes to translate the user's persistent objects into DSPACK or database objects, which are then read or written by the underlying I/O system. Refer to Appendix B for more information about DØOM's dictionary classes.

1.2.1 Undefined Objects and Schema Evolution

The definition of DØOM persistent classes is fixed at compilation time (actually at preprocessing time). The universe of persistent classes that is known to a particular program is fixed at link time. Each persistent class known to DØOM has a name that is deterministically mapped onto a single DSPACK data set or database table. One DSPACK data set or database table stores data from all instances of a particular persistent class.

Ideally, when DØOM reads previously written data, the names and definitions stored in the previously written data correspond exactly to the current DØOM persistent classes. In such cases the conversion of DSPACK data into DØOM objects is straightforward. But if the DSPACK definitions do not match the compiled-in definitions, DØOM will attempt to convert the data format. DØOM can cope with simple situations such as the addition, deletion, or rearrangement of data fields. Any missing data is set to zero or null. If DØOM can't convert the data format, a fatal error results. For more discussion on schema evolution, see Section 8.

Table 1: DØOM atomic types.

DØOM type	Equivalent standard type
d0_Int	int
d0_Uint	unsigned int
d0_Short	short
d0_UShort	unsigned short
d0_Long	long
d0_ULong	unsigned long
d0_Char	unsigned char
d0_Bool	bool
d0_Float	float
d0_Double	double
d0_Octet	unsigned char

It can also happen that DØOM finds a DSPACK data set or database table for which there is no corresponding persistent class (i.e. an unknown name). This situation is described in Section 7.

2 DØOM Object Model Classes and Types

This section contains a more detailed description of the DØOM object model classes and types.

2.1 Atomic Types

A complete list of DØOM atomic types is shown in Table 1. DØOM atomic types are implemented as simple typedefs. Persistent class headers may be specified using either the DØOM name or the standard name for the allowed atomic types.

Note that DSPACK does not support 64-bit integers. If you use a `long` type on the Alpha, it will be truncated to 32 bits on output.

2.2 Container Classes

Many types of containers can be saved. A container is represented as a one-dimensional ordered list of arbitrary length of homogeneous objects.

The type of object stored in a container should satisfy the same requirements as for a class data member. Atomic types, pointers, references, user-defined classes, and other containers are all legitimate.

Polymorphism is not allowed for objects stored in containers. The type of any object put into a container must exactly match the type of the class's template argument. It follows that heterogeneous collections are not allowed. However, the same effect as a heterogeneous collection can be achieved by having a collection of references or pointers.

Note that some container classes can have additional state besides the container contents, such as the comparison objects for associative containers, or the allocator object for most STL containers. This additional state is *not* saved. You should thus be careful if you create a container with a comparison object or allocator which is not simply initialized by the default constructor.

The kinds of containers which may be saved are summarized below.

2.2.1 STL Container Classes

The following plain STL container classes can be saved:

- `deque`
- `list`
- `vector`
- `set`
- `multiset`
- `map`
- `multimap`
- `stack`
- `queue`
- `priority_queue`
- `valarray`

2.2.2 Hash Table Classes

The following hash table classes, defined in the `d0_util` package, can be saved:

- `d0_util::Hashmap`
- `d0_util::Ptrmap`
- `d0_util::Ptrset`
- `d0_util::Strptrmap`
- `d0_util::Stringset`

Table 2: DØOM and STL container classes.

DØOM	STL	Description
`d0_Vector<T>`	`vector<T>`	Variable length contiguous array
`d0_List<T>`	`list<T>`	Linked list
	`set<T, less<T> >`	Unordered collection (no duplicates)
	`multiset<T, less<T> >`	Unordered collection (duplicates allowed)
	`map<K, T, less<K> >`	Map (no duplicates)
	`multimap<K, T, less<K> >`	Map (duplicates allowed)

2.2.3 Other C++ Types

The following additional template classes from the standard C++ library may be saved:

- `bitset`
- `complex`

2.2.4 DØOM Container Classes

[Note: The classes described in this section are now in the `d0_util` package.]

The DØOM container classes are derived from STL containers. Table 2 table summarizes the correspondence between DØOM and STL container classes. The DØOM containers also inherit a common ODMG-like interface from a generic container class `d0_Collection<T>` (see Fig. 1). The ODMG-like interface is a partial implementation of the interface defined in Ref. [2]. Users can use either the STL interface or the ODMG-like interface. Note that the DØOM containers are not derived from `d0_Object`, and therefore can not be persistent on their own. This does imply, however, that these containers can be used apart from the rest of the persistence mechanism.

Classes that are to be stored in DØOM containers should implement the default constructor, copy constructor, destructor, and any methods required by the corresponding STL container. These typically include the assignment operator (`=`), equivalence operator (`==`), and, if appropriate, the less than operator (`<`). The other relational operators are derived from global template functions that are part of STL. As with any class, it is also good practice to implement the `ostream` insertion operator (`<<`).

Iterators DØOM contains two iterator classes, called `d0_Const_Iterator<T>` and `d0_Iterator<T>`, for iterating over elements of any of the DØOM containers. The

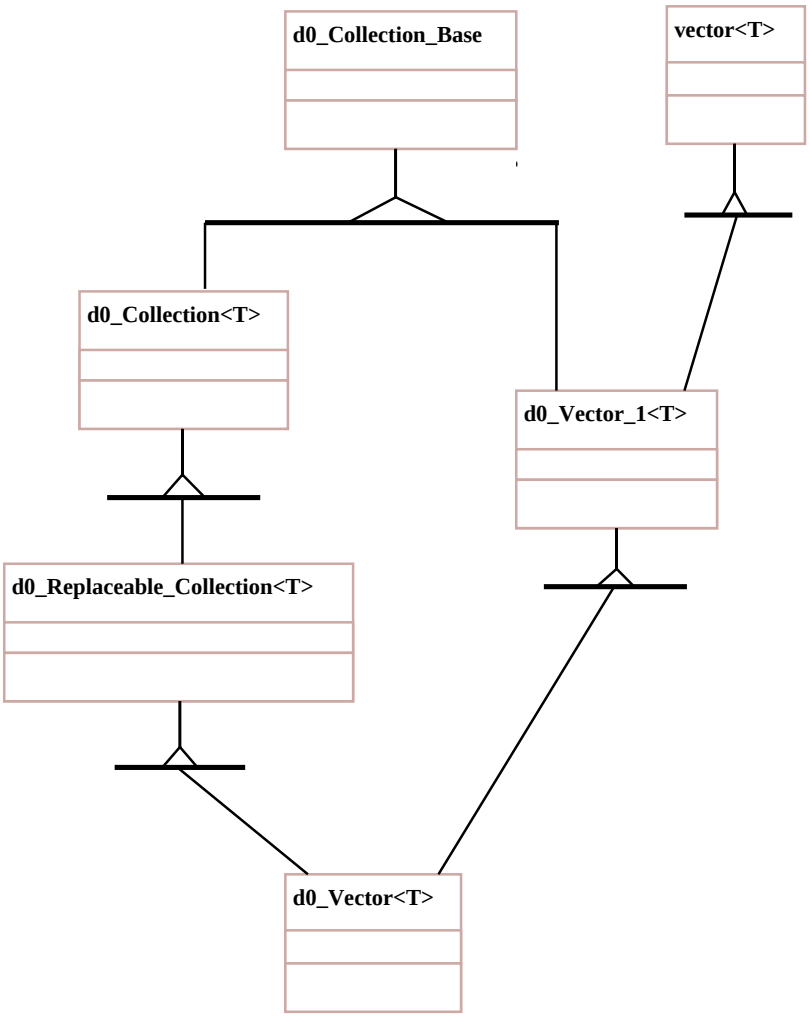

Figure 1: Mutable container (`d0_Vector` and `d0_List`) class diagram.

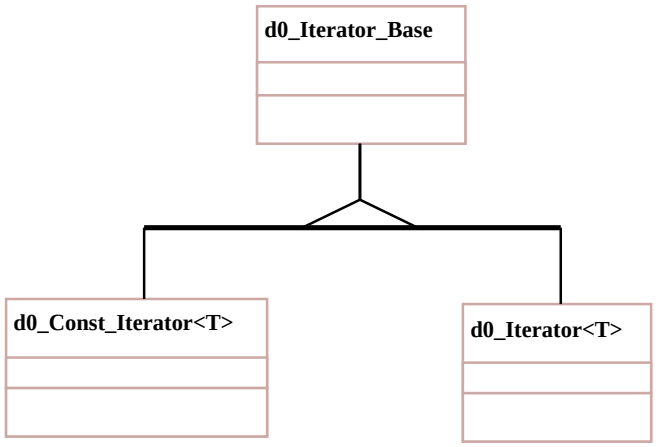

Figure 2: Iterator class diagram.

Table 3: Pointer and Reference container classes.

Special class	Equivalent ordinary class
d0_PVector<T>	d0_Vector<T*>
d0_RVector<T>	d0_Vector<d0_Ref<T> >
d0_PList<T>	d0_List<T*>
d0_RList<T>	d0_List<d0_Ref<T> >
d0_PIterator<T>	d0_Iterator<T*>
d0_RIterator<T>	d0_Iterator<d0_Ref<T> >
d0_Const_PIterator<T>	d0_Const_Iterator<T*>
d0_Const_RIterator<T>	d0_Const_Iterator<d0_Ref<T> >

constant version of the iterator must be used for read only collections and for immutable collections (sets and multisets). The class diagram for the DØOM iterators is shown in Fig. 2. There is no inheritance relationship between the DØOM iterators and STL iterators. However, DØOM iterators fulfill the interface requirements for STL bidirectional iterators, and therefore may be used wherever STL iterators are allowed, for example in STL algorithm calls. In addition to the STL interface, DØOM iterators have an ODMG-like interface that provides some additional functionality not present in the STL interface.

DØOM iterators can be obtained from any DØOM container class using the methods `d0_begin()` and `d0_end()`. The STL methods `begin()` and `end()` return STL iterators, which can also be used for iterating over DØOM collections.

Pointer and Reference Container Classes Special container and iterator classes exist for collections of pointers and references. Table 3 lists these special classes and their functional equivalents. These classes exist for the technical reason that they allow more sharing of code than the ordinary classes, and hence they produce smaller code. Either kind of special collection can be used in persistent classes.

2.2.5 User-Defined Collections

It is also possible to make DØOM treat an arbitrary class as a collection. For this to make sense, it must be possible to represent the state of an instance of the class by an ordered list of values of some uniform type. There are several steps required to use a user-defined collection type:

1. You must define an *adapter* class for your collection class. This makes it possible to get data into and out of the collection using a uniform interface. Your adapter class must derive from `d0om_Collection_Adapter` (see `d0om/d0om_Collection_Adapter.hpp`). It must implement the virtual methods `size`, `collsize`, `insert_elements`, `iterate`, `construct`, and `destroy`.

   Note that there several existing adapter classes which you might be able to use.

   - If your collection has an interface compatible with an STL sequence, you can use `d0om_STL_Sequence_Adapter`. Specifically, it must:
     - Define `value_type`, which must have a default constructor.
     - Define `iterator` as at least a forward iterator.
     - Implement `clear()`.
     - Implement `size()`.
     - Implement `begin()`.
     - Implement `end()`.
     - Implement `insert (p, n, t)`, where `p` is an `iterator`, `n` is an integer, and `t` is an instance of `value_type`.
   - If your collection has an interface compatible with an STL associative container, you can use `d0om_STL_Assoc_Adapter`. Specifically, it must:
     - Define `value_type`, which must have a default constructor.
     - Define `iterator` as at least an input iterator.
     - Implement `clear()`.
     - Implement `size()`.
     - Implement `begin()`.
     - Implement `end()`.
     - Implement `insert (p, t)`, where `p` is an `iterator` and `t` is an instance of `value_type`.

2. Your collection should define the `value_type`, giving the type of a collection element.

3. Your collection should also define `d0om_collection_adapter` as a typedef name for the adapter class. Note that this name must be defined in the collection class itself, and not in a base class (otherwise, it would be impossible to derive an ordinary class from a collection class).

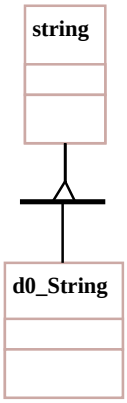

Figure 3: String class diagram.

4. In most contexts where you use your collection class, the complete declaration of the adapter class is not required. However, it is required when you compile linkage files for classes which use the collection. For the adapter classes listed above, this is handled automatically. Otherwise, you must arrange this yourself. This can be done by including in the header defining the collection class the construction

   ```
   #ifdef __D0CINT__
   # pragma linkageinclude "adapter-header"
   #endif
   ```

   This causes `d0cint` to emit `#include "`*adapter-header*`"` in the generated linkage file.

For an example of this, see `mycoll`, `mycoll_adapter`, and `myclasses10` in the DØOM tests directory.

If you are want to adapt an existing class without modifying it but are having trouble due to the required typedefs, the `#pragma extendclass` directive may be useful. See Section 9.1 for more information.

2.3 DØOM String Class

[Note: The `d0_String` class is now in the `d0_util` package.]

The DØOM string class is derived from the ANSI standard string class (see Fig. 3). The DØOM string class exists for various technical reasons: it fixes bugs and omissions in various vendor's `string` implementations, and it can lead to much shorter external names. (This has been seen to make a considerable difference in object file sizes.) From the user's point of view, `d0_String` provides the same functionality as `string`. The GNU C++ compiler (v2.7) is known to contain a bug for which the workaround is to include the DØOM string header before any other DØOM header.

The plain C++ `string` (or `basic_string`) class can also be saved. Also note that the `d0_String` class is independent from the rest of the persistency mechanism.

2.4 DØOM Reference Classes

The DØOM reference class `d0_Ref<T>` functions in many ways like the pointer type `T*`. In particular, `d0_Ref<T>` can be dereferenced by the C++ operators `*` and `->`. The class `d0_Ref<T>` is sometimes called a "smart pointer." References provide more functionality than standard pointers. DØOM references currently have the following features.

1. Reference counting of dynamic C++ objects.
2. Deferred conversion of DSPACK objects to C++ format.
3. Deferred I/O for database objects.

2.4.1 Reference Counting

A reference count is logically associated to every persistent object on the heap. This is true whether or not an object has been instantiated in C++ format. For efficiency reasons, whenever possible DØOM defers I/O or data conversion until a reference has been dereferenced. References to static and automatic objects do not have reference counts, but such objects can still be pointed to by `d0_Ref<T>`. Object reference counts interact with DØOM references as follows:

1. Objects created using the `new` operator get a reference count of zero.
2. When an object is read from external storage, a reference pointing to a non-instantiated C++ object, and having a reference count of one, is returned to the user.
3. The reference count is incremented when a reference is set to point at the object. This normally happens when a reference is initialized, assigned, or copied.
4. When a reference is cleared or destroyed, the reference count associated with the pointed-to object is decremented. If the reference count reaches zero, the object is deleted.

The programmer is not responsible for deleting objects that are managed by references. Because the reference itself takes care of deleting heap objects, memory leaks and dangling references are less likely than with standard pointers. Memory leaks are still possible if several objects containing references point to each other in a loop. In such cases, the programmer must break the loop in at least one place by calling the `clear()` method of `d0_Ref<T>`.

It is possible to create a `d0_Ref` from a C++ pointer without affecting the reference count of the C++ object. This is done with the following construction:

```
X* xptr;
...
d0_Ref<X> xref (xptr, d0_Ref_Base::NOREFCOUNT);
```

This will not affect the reference count of the object pointed to by XPTR, either when the reference is constructed or when it is deleted. This property is copied along with the reference.

2.4.2 `d0_Ref<T>` Methods

The following methods are defined for `d0_Ref<T>`:

`d0_Ref()`	Constructor.
`d0_Ref(T*)`	Constructor.
`d0_Ref(d0_Ref<U>)`	Constructor.
`d0_Ref(T*, Norefcount)`	Constructor that doesn't alter the reference count.
`T* operator->()`	Dereference.
`T& operator*()`	Dereference.
`d0_Ref<T>& operator= (T*)`	Assignment.
`d0_Ref<T>& operator= (d0_Ref<U>&)`	Assignment.
`operator bool ()`	Test for a non-null reference.
`bool operator!()`	Test for a null reference.
`void clear()`	Clear the pointer and decrement the reference count.
`const d0om_Type_Object* d0om_type()`	Return the dynamic type of the object which this `d0_Ref` references. If possible, do it without creating the object. Whether this is possible or not depends on the I/O backend.
`void delete_object()`	Clear the pointer and delete the pointed-to object. Throw an exception if the reference count is greater than one.
`bool has_field (const d0_String& fieldname, const d0om_Type_Class* cls = 0)`	Examine the version of `cls` present in the same place that the object that this `d0_Ref` points to is located. (In the same DSPACK event, for example.) If `cls` is defaulted to zero, examine the class of the object that this reference points to. Return true if that version of `cls` had a field named `fieldname`. (I.e., return true if the field named was really present in the data being read, rather than being defaulted.)

	At present, this only really works with the DSPACK backend; others will just test the for `fieldname` in the current version of the class. If possible, we do this operation without actually creating the object to which this `d0_Ref` points; whether this is possible or not depends on the I/O backend.
`bool is_deferred()`	True if the object at which this reference is pointing has not yet been constructed.
`bool is_dereferenceable()`	True if it is safe to dereference this `d0_Ref`. (I.e., not `is_null` and either not `is_unknown` or `d0om_Options::unknown_action` has been set to `MAKE_UNKNOWN`.)
`bool is_null()`	Is anything being pointed to?
`bool is_unknown()`	True if the object at which this reference is pointing does not have compiled-in type information. (See Section 7.)
`T* ptr()`	Return the corresponding C++ pointer.
`T* ptr_only()`	Like `ptr()`, but if the C++ object hasn't yet been constructed, just return null.
`void purge_object()`	Delete the object instance this `d0_Ref` is pointing at, but leave the reference valid so that the object can be recreated from persistent storage if it is dereferenced again. Exceptions: If the `d0_Ref` isn't doing reference counting, this just clears it. If the `d0_Ref` is pointing at an object that wasn't read from persistent storage, the `d0_Ref` is just cleared. If there's more than one `d0_Ref` pointing at the object, an exception is thrown.
`T* release()`	Release ownership of the object (similar to `auto_ptr<T>::release`). If the object this `d0_Ref` is pointing to is not reference-counted, just return the pointer to it. Otherwise, if the reference count is not 1, throw an exception. Otherwise, clear this reference, reset the reference count on the object to 0, and return the object pointer.

```
int version (const d0om_Type_Class* cls = 0)
```

Examine the version of `cls` present in the same place that the object that this `d0_Ref` points to is located. (In the same DSPACK event, for example.) If `cls` is defaulted to zero, examine the class of the object that this reference points to. Return the version number of that instance, or 0 if we couldn't find a version number. For backends that don't store version information, this will return the current version. If possible, we do this operation without actually creating the object to which this `d0_Ref` points; whether this is possible or not depends on the I/O backend.

All six relational operators are defined for `d0_Ref`. A `d0_Ref` can also be compared for equality with bare pointers.

The class `d0_Ref` also provides the typedef names `pointer` (`T*`), `const_pointer` (`T const *`) and `element_type` (T).

The following global typedefs are provided for convenience:

```
d0_Ref_Any        d0_Ref<d0_Object>
d0_Const_Ref_Any  d0_Ref<const d0_Object>
```

The `has_field` and `version` methods deserve a little more explaination for the `cls` argument. DØOM can only handle references (and pointers) to instances of `d0_Object`. Thus, if one wants to specify an instance to DØOM it must be an instance of `d0_Object`. But for these methods, one would like to be able to get information about classes that do not derive from `d0_Object`. That is why the `cls` argument is present. For example, given the following definitions:

```
struct A {};
struct B :
  public d0_Object
{
  D0_OBJECT_SETUP (B);
  A a;
};

d0_Ref<B> r;
```

Then `r->version()` will give the version of `B` in the file from which the object pointed to by `r` was read. To get the version of `A`, use `r->version (A::d0om_type_static())`.

Also, when using `has_field`, note that DØOM does not consider a class to directly contain fields from base classes. Thus, if in addition to the above definitions you had

```
struct C
  : public B
```

```
{
  D0_OBJECT_SETUP (C);
};
d0_Ref<C> rc;
```

then `rc->has_field ("a")` will return `false`, but the construction `rc->has_field ("a", B::d0om_static_class())` will return `true`.

2.4.3 Restrictions

Classes `T` pointed to by `d0_Ref<T>` must directly or indirectly derive from `d0_Object`, regardless of whether or not they are intended to be persistent.

Classes pointed to by `d0_Ref<T>` are allowed to be polymorphic or abstract.

2.4.4 Name-Only References

C++ allows pointers to be declared that point to incompletely defined classes. The same is true of DØOM references.

```
class T;
d0_Ref<T> pT;
```

Such name-only reference declarations are desirable in order to reduce the physical coupling between modules. Some restrictions apply to name-only references. The following usages are allowed for name-only references.

1. Declaration
2. Copying
3. `clear()`, `delete_object()` and `is_null()` methods.

The following usages are disallowed for name-only references.

1. Dereferencing.
2. Initialization from a C++ pointer.
3. `ptr()` method.

2.4.5 Reference Type Conversions

Default pointer conversions are carried out automatically when assigning between `d0_Ref` instances. For non-default conversions, you must use an explicit cast. The syntax is as follows:

`D0_REFCAST((`*type*`))(`*object*`)`	Downcast
`D0_REF_UNSAFECAST((`*type*`))(`*object*`)`	Unsafe cast
`D0_REFCAST((const `*type*`))(`*object*`)`	Constant downcast
`D0_REF_UNSAFECAST((const `*type*`))(`*object*`)`	Constant unsafe cast

`D0_REFCAST` acts like a `dynamic_cast`, except that it will generate a fatal error if the type of the object is not valid. `D0_REF_UNSAFECAST` does not do any checking; it is not recommended for general use.

If you are using a compiler that supports explicit specification of function template arguments (i.e., everyone except Microsoft), you can also use the notations

`d0_refcast<`*type*`>(`*object*`)`	Downcast
`d0_ref_unsafecast<`*type*`>(`*object*`)`	Unsafe cast

Here are some examples:

```
d0_Ref<myclass> r1 = new myclass;
d0_Ref_Any r2 = r1;
d0_Ref<myclass> r3 = D0_REFCAST((myclass)) (r2);
d0_Ref<const myclass> r4 = r1;
d0_Ref<const myclass> r5 = D0_REFCAST((const myclass)) (r2);
```

The above macros are type-safe in the sense that a prohibited conversion will result in a compiler error. Refer to the header file `d0_Ref.hpp` for more information about reference type conversions and how to use the above macros.

Note that these macros will not work with type names containing commas. You must define a separate typedef name for those. The old macros `D0_REFCONV`, `D0_CONST_REFCONV`, `D0_CONST_REFCAST`, and `D0_REF_CONST_UNSAFECAST` are no longer needed, but are retained for backwards compatibility.

2.5 C++ Bare Pointers

C++ bare pointers are allowed in persistent classes. The pointed-to class must derive from `d0_Object`. Pointers are allowed mainly so that preexisting classes can more easily be made persistent. For new classes, it is recommended to use the DØOM reference class `d0_Ref<T>`, as described in section 2.4.

Whenever an object with C++ pointers is read, all the objects to which it points are also read (no deferred I/O). The user is responsible for deleting these objects when done with them.

Because of the different way in which references and pointers manage memory, pointers and references should not simultaneously be made to point to the same object. DØOM attempts to enforce this prohibition.

As with DØOM references, persistent pointers can be used to point to polymorphic or abstract classes.

There are two functions available to do the equivalent of `d0_Ref::version` and `d0_Ref::has_field` for bare pointers:

```
namespace d0om {

  int get_version (const d0_Object* o, const d0om_Type_Class* cls = 0);

  bool has_field (const d0_Object* o,
```

```
                    const d0_String& fieldname,
                    const d0om_Type_Class* cls = 0);
}
```

They work just like the `d0_Ref` methods described in section 2.4.2, except that they take a pointer to the object as an addtional argument. These functions get declared if you include the header `d0_Ref.hpp`.

2.6 C++ `auto_ptr` Class

The C++ `auto_ptr` class may also be used, subject to the same restrictions as for bare C++ pointers. In addition, it is an error to write a structure which contains more than one `auto_ptr` points at a given object, though there is no checking for this. (Be careful with using the CD2 `auto_ptr` class, as it is easy to leave unowned dangling pointers. The version of `auto_ptr` actually approved for the standard should make it harder to run into this.)

2.7 Transient Data in Persistent Classes

Classes are allowed to contain transient data members. Such members are not read to or written from the persistent store. A class designer can mark a data member transient using the directive `#pragma transient` *member-names*. Here, *member-names* is a comma-separated list of the names of the members which you want to be transient. They are looked up using the scope which is current at the point where the directive appears; the directive does not have to be lexically inside the class which it is modifying. A `#pragma transient` directive should probably be placed inside a `#ifdef __D0CINT__` construction, to hide it from translators other than `d0cint`. (Or use `D0OM_TRANSIENT`; see section 9.2.) Alternatively, a field may be marked as transient by including a C++ comment beginning with the string "`//!`" on the same line as the field declaration. (This convention was borrowed from ROOT.) This syntax, however, is deprecated. Also, any data field that is not a recognized DØOM persistent type is transient by default (but you may get a warning).

Examples:

```
class foo
{
public:
  int a;
  int b;  //! foo::b is transient

  struct bar
  {
    int d;
    int e;
  };
```

```
  int f;

#ifdef __D0CINT__
# pragma transient f // foo::f is transient
#endif
};


#ifdef __D0CINT__
# pragma transient foo::a, foo::bar::e
  // foo::a and foo::bar::e are transient.
#endif
```

The `d0_Object` interface provides two methods, `activate` and `deactivate`, for initializing and deinitializing transient members when they are moved into and out of memory (see section 2.8.3).

Related to `#pragma transient` is `#pragma nowrite`, which declares that a member is to be read, but not written. See Section 8.4 for further discussion.

2.8 DØOM Base Class and Inheritance

A persistent class which is to be referred to by a pointer or `d0_Ref` must inherit, directly or indirectly, from the persistent base class `d0_Object`. Either single or multiple inheritance may be used. Figure 4 shows an example of persistence via single inheritance.

```
class A : public d0_Object { ... };
class B : public A { ... };
```

The classes `A` and `B` are both persistent.

If a class is used only as a data member of another class or as the element type of a container, then it need not derive from `d0_Object`. Example:

```
class A {};

class B : public d0_Object
{
public:
  A a;  // OK
  std::list<A> alist;  // OK
  A* aptr;  // BAD
  d0_Ref<A> aref;  // BAD
  B* bptr;  // OK

  D0_OBJECT_SETUP (B);
}
```

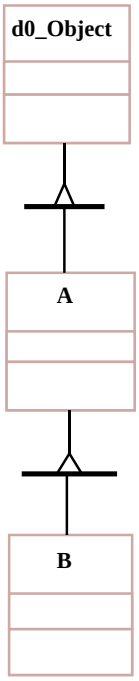

Figure 4: Persistence via single inheritance.

2.8.1 Multiple Inheritance

Figure 5 shows the simplest case of persistence via multiple inheritance.

```
class pA : public A, public d0_Object { ... };
```

Multiple inheritance has been used to define a persistent version `pA` of a non-persistent class `A`. Note that only persistence capable fields of `A` will persist in `pA`.

A more complex example of multiple inheritance is shown in Figure 6.

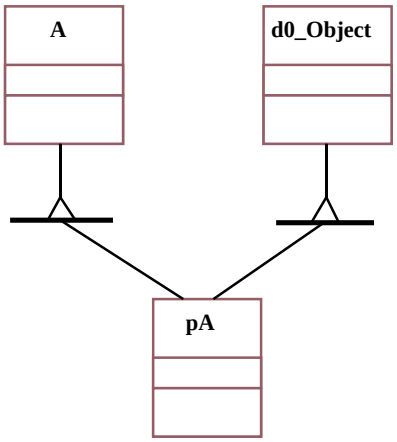

Figure 5: Persistence via multiple inheritance.

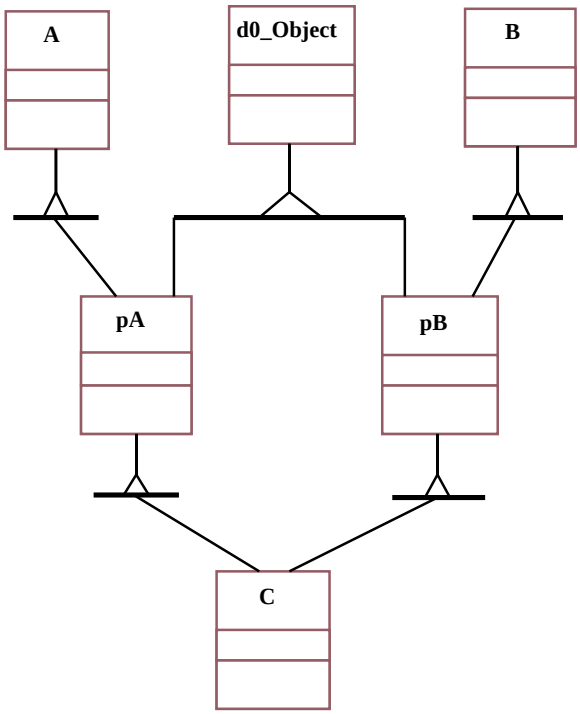

Figure 6: Persistence via multiple virtual inheritance.

```
class pA : public A, virtual public d0_Object { ... };
class pB : public B, virtual public d0_Object { ... };
class C : public pA, public pB { ... };
```

In this example, class `C` multiply inherits from two persistent classes `pA` and `pB`. The classes `pA` and `pB` must use virtual inheritance with respect to `d0_Object` to ensure that only copy of `d0_Object` is contained in class `C`. In general, the following restrictions apply to the use of multiple inheritance.

1. Only one copy of any class may be inherited in a persistent class. Use of virtual inheritance may be necessary to ensure this.

2. Virtual base classes may not contain any persistent data.

2.8.2 Polymorphism

Polymorphism is allowed in the DØOM object model via the reference class `d0_Ref<T>` and pointers. References and pointers in persistent classes can point to polymorphic or abstract classes.

2.8.3 Activate and Deactivate

In general, a persistent class designer does not need to be concerned very much with the interface of the class `d0_Object`. Possible exceptions are the methods

"`void activate()`" and "`void deactivate()`" of `d0_Object`. The `activate()` method is called whenever a persistent object is instantiated in memory (all persistent fields should be filled in at this point). The `deactivate()` method is called just before an object is written. `activate()` is intended to regenerate transient data that is dependent on persistent data within the same object. Note that this task can not be handled by a constructor because the constructor does not have access to the object's persistent data. `deactivate()` does the same process in reverse.

A class designer may override `activate()` or `deactivate()` if they are needed for nontrivial processing. In that case, `activate()` and `deactivate()` should explicitly invoke the `activate()` and `deactivate()` methods in any base classes.

Note that `deactivate()` is declared `const`, and thus any overriding function must also be declared `const`. This is a statement of the constraint that `deactivate()` should not change the state of the object as it appears from the outside. If certain class members do need to be changed, they should probably be declared `mutable`.

2.9 Class Versions

User-defined classes can be defined with an integer version number. When an instance of the class is read in, you can query it to see with what version it was written, through either the `d0_Ref<T>::get_version` method or the `d0om::get_version` function (see sections 2.4.2 and 2.5). If the version number is smaller than the current versions, this can also trigger a user-written conversion on input (see section 8.3.3).

You specify the version of a class to DØOM using the `#pragma version` directive:

```
class A {};
#ifdef __D0CINT__
#pragma version A 5;
#endif
```

The version number may be an expression that evaluates to a constant integer. The directive should probably be within a `#ifdef __D0CINT__` construct to hide it from processors other than `d0cint`. (Or use `D0OM_VERSION`; see section 9.2.)

The version number is stored in the data file as part of the dictionary information. Thus, it does not take up space in each instance of the class. At present (`v00-25-00`), only the DSPACK supports storing version numbers.

2.10 How Objects Are Reconstructed

When an object is being read in, it is first initialized using one of its constructors. Next, all persistent fields are filled in. (No class methods are invoked for this.) Next, if the object's class derives from `d0_Object`, the `activate` method is called. Finally, if the object is being stored in an associative container, the object's copy constructor is used to copy it to its final position.

Normally, the default constructor is used to initialize the object. However, sometimes the default constructor does stuff which is not appropriate for when an object

is being read. For example, if an object has a persistent bare C++ pointer, and the default constructor initializes it to newly-allocated memory, then that memory would be leaked when the object gets read in. (Note, however, that this particular problem would go away if a `d0_Ref` were used instead of a bare pointer.)

To solve this problem, you can define a constructor which has a signature of '`(const d0_Input_Info*)`'. If such a definition is present, then that constructor will be used instead of the default constructor to initialize objects which are being read in. Presently, the pointer which gets passed to the constructor will be null, but it may be used in the future to provide additional information to the constructor.

If a `d0_Input_Info` constructor is provided, the default constructor need not be present.

Note also that this behavior is only applicable for top-level (complete) objects. For objects contained inside of other objects, the constructor used is determined by the containing object.

2.11 Nested Classes

Nested classes (classes defined within other classes) are allowed in DØOM. Nested classes can be persistent or contained in other persistent classes. The rules for persistent nested classes are the same as for non-nested classes. Namely, they must derive from `d0_Object` and otherwise conform to the DØOM object model. Note that nested classes or structures must be named; they cannot be left anonymous. Also, if a nested class derives from `d0_Object`, it must be public.

2.12 Namespaces

Namespaces may be used. A name in a namespace is treated much like a name in a class scope. The `using` directive is not fully implemented; you should explicitly qualify any names you use in a header rather than using a `using` directive.

2.13 Preprocessor Macros

`Cint` has only a limited implementation of the C preprocessor.

Conditionals on macro definitions (`#if defined`, `#ifdef`, `#ifndef`) should work.

In most cases, simple macros (without parameters) should work as long as the macro definition is a constant or a type name.

Macros with arguments can be used, but only in certain contexts. They should work at the start of a statement, or in a position where a function call may appear. They probably won't work in other places. Token pasting works, but stringification does not.

In general, it is a good idea to rely on preprocessor macros as little as possible in sources which `d0cint` is supposed to read.

2.14 Template Classes

Template classes are allowed in DØOM. Template classes can be persistent or contained in other persistent classes. The rules for persistent template classes are the same as for non-template classes. Namely, they must derive from `d0_Object` and conform to the DØOM object model. In addition it is necessary to declare to the DØOM preprocessor those instantiations of a template class that are to be made persistent. This is done using the `#pragma linkage` directive (see Section 5.9):

```
#pragma linkage my_template_class<int>
```

These directives should usually be put inside an `#ifdef __D0CINT__` guard to hide them from translators other than `d0cint`. (Or use `DOOM_LINKAGE`; see section 9.2.) The name given in the directive is looked up in the scope which is current at the point where it occurs.

A template instantiation may also be declared with a pseudocomment with the following format:

```
//! +class my_template_class<int>
```

In this case, the name must be fully qualified, i.e., if it is in a namespace or class scope, this must be given explicitly.

The linkage directive is probably best placed where where the instantiation is used.

Here is an example:

```
/*** tmpl.hpp ***/

template <class T>
struct tmpl
{
  T x;
};



/*** tint.hpp ***/

#include "tmpl.hpp"
#include "d0om/d0_Object.hpp"

class tint : public d0_Object
{
public:
  tmpl<int> y;
};
```

```
#ifdef __D0CINT__
# pragma linkage tmpl<int>
#endif



/*** tfloat.hpp ***/

#include "tmpl.hpp"
#include "d0om/d0_Object.hpp"

class tfloat : public d0_Object
{
public:
  tmpl<float> y;
};

#ifdef __D0CINT__
# pragma linkage tmpl<float>
#endif
```

One would then run `tint.hpp` and `tfloat.hpp` through `d0cint`. It is unnecessary to run `d0cint` on `tmpl.hpp` here, as that header does not actually define any classes — it only defines the template.

An alternate method for signaling that `d0cint` should generate linkage information for a template class is provided by the `d0om_autolink` typedef. This is described in Section 5.9.

It is also possible to use non-type template arguments; however, `d0cint`'s support for them is incomplete. Numeric types should work ok. The character string types `char*` and `const char*` should also work, with the caveat that `d0cint` will not attempt to evaluate expressions given for such arguments — the expression text is simply copied. This implies that any nested names appearing as a character string template argument should be fully namespace- and class-qualified.

Types other than the above are not really supported; they may work sometimes, but that's more by chance than by design...

2.15 Translated Classes

Sometimes, one wants to use a class from an external source. This class does not satisfy the requirements of DØOM, and it cannot be changed, but you want to be able to save it. In some cases, this can be accomplished by defining it as a *translated* class, as described below.

Note that as this mechanism is somewhat complicated to set up, we recommend that it be used mainly to adapt existing external classes. New DØ code should probably be written to use DØOM directly. In addition, the class to be translated can only be used as a member of another class — you can't have a pointer to it directly. (However, the translated class can have internal pieces which are reached by pointers, as long as these pieces are not directly referenced by DØOM objects.)

Setting up a translated class actually requires three classes. The first is the class you want to adapt, called the *target* class. Given the target class, you should write another class which contains all the persistent state of the target class and which satisfies the requirements of DØOM. This is called the *dummy* class. Finally, you need to write the *translator* class, which knows how to convert between the target class and the dummy class. The translator class should derive from the abstract class `d0om_Class_Translator`. Here is a list of methods which it should implement.

- `void* makedum_fromtarg (const void* targ) const`
  Given a pointer to an instance of the target class, create an instance of the dummy class containing the same information.

- `void  deldum (void* dum) const`
  Delete an instance of the dummy class (which was created by the method `makedum_fromtarg`).

- `void* makedum_empty () const`
  Make an empty instance of the dummy class.

- `void copydum_totarg (void* targ, void* dum) const`
  Given a pointer to an instance of the dummy class (created by `makedum_empty`) and an instance of the target class, copy the dummy instance to the target instance. Then delete the dummy instance.

- `void construct (void* targ) const`
  Construct an instance of the target class.

- `void destroy (void* targ) const`
  Destroy an instance of the target class.

- `void zero (void* targ) const`
  Clear an instance of the target class.

- `int size () const`
  Return the size of an instance of the target class, in bytes.

- `int align () const`
  Return the required alignment of an instance of the target class, in bytes.

The translator class is linked to the dummy class by putting a typedef declaration of `d0om_class_translator` in the dummy class.

There is a also simplified translator interface that can be used in many circumstances. To use it, both the target and dummy classes must have default constructors, and it should be appropriate to align instances of the target class at eight-byte boundaries. In that case, you can instead derive from the template class `d0om::Translator_Helper<Target, Dummy>`, where `Target` and `Dummy` are the target and dummy classes, respectively. (This class in turn derives from `d0om_Class_Translator`). The `d0om::Translator_Helper` class has only three pure methods that must be supplied:

- `void zero_target (Target& targ) const`
  Clear out the target instance `targ`.
- `void target_to_dummy (const Target& targ, Dummy& dum) const`
  Convert `targ` to `dum`.
- `void dummy_to_target (const Dummy& dum, Target& targ) const`
  Convert `dum` to `targ`.

How this all gets organized in the header files is a bit tricky. I'll go through an example, showing one way which seems to work.

Suppose you want to adapt a class named `C`. The header file for this class is in `otherstuff/C.hpp`. The package in which you're including the translator is called `mystuff`.

First, create the dummy class. That can be defined in the file `mystuff/C_dum.hpp`, and look something like this:

```
class C_Translator;

#ifdef __D0CINT__
class C
#else
class C_dum
#endif
{
public:
  typedef C_Translator d0om_class_translator;

  // Define the persistent data here.
  ...
};


#ifdef D0OM_LINKAGE_FILE
typedef C_dum C;
#endif
```

```
#ifdef __D0CINT__
# pragma linkageinclude "mystuff/C_Translator.hpp"
#endif
```

When this header is included in normal C++ code (such as from the translator class), it defines the dummy class `C_dum`. However, when run though `d0cint`, it appears to define the class `C`. This is the definition of class `C`, as far as `d0cint` is concerned. The linkage file which `d0cint` emits will then contain references directly to the class `C`. That's the reason for the line `typedef C_dum C`. This line is visible only when the linkage file is being compiled. You tell `d0cint` that this is a translated class with the `d0om_class_translator` typedef; the target of the typedef is the translator class. In order for the generated linkage file to compile, the full declaration of the translator class must be present. But it may not be desirable to couple `C_dum.hpp` to `C_Translator.hpp`. This can be avoided by using the `linkageinclude` directive, as in the example. The directive `#pragma linkageinclude` *text* causes `d0cint` to emit `#include` *text* in the generated linkage file.

This header defining the dummy class should be run through `d0cint`, generating a linkage file and a reference header.

Next, write the translator class. In this example, it would be called `C_Translator` and be defined in `C_Translator.hpp`. Finally, you may need a wrapper for the target class. This can be called `mystuff/C.hpp`, and might look something like

```
// If d0cint will read otherstuff/C.hpp, you can just include it here.
#include "otherstuff/C.hpp"

#ifdef __D0CINT__
#pragma linkageinclude "mystuff/C_dum_ref.hpp"
#endif
```

The `linkageinclude` here ensures that any persistent class using this header will automatically get the definition for the dummy class. If `d0cint` will not read `otherstuff/C.hpp`, you can instead do the following:

```
#ifdef __D0CINT__
  // A dummy definition of C, that d0cint can parse.
  ...
#else
# include "otherstuff/C.hpp"
#endif
```

Then, to use the translated class, you include `mystuff/C.hpp` from your classes, rather than `otherstuff/C.hpp`.

Note that, at present, this mechanism is available only if you are using the DSPACK backend.

2.16 ZOOM and CLHEP Classes

The following classes from Zoom and CLHEP may also be used:

- `LinearAlgebra`
  - `MatrixC`
  - `MatrixD`
  - `ColumnVector`
  - `RowVector`
- `PhysicsVectors`
  - `SpaceVector`
  - `UnitVector`
  - `LorentzVector`
  - `PlaneVector`
  - `AxisAngle`
  - `EulerAngles`
  - `Rotation`
  - `RotationX`
  - `RotationY`
  - `RotationZ`
  - `LorentzTransformation`
  - `LorentzBoost`
  - `LorentzBoostX`
  - `LorentzBoostY`
  - `LorentzBoostZ`
- `CLHEP/Vectors`
  - `Hep3Vector`
  - `HepLorentzVector`
  - `HepRotation`
  - `HepLorentzRotation`
- `CLHEP/Matrix`
  - `HepMatrix`
  - `HepDiagMatrix`

- `HepSymMatrix`
- `HepVector`

The `d0om_zm` package must be accessible for this to work. If you use any of these classes in a persistent class, you should link with the additional library `-ld0om_zm`.

The names from the `FixedTypes` header may also be used.

Note that some caution is needed with the vector classes from CLHEP, as there are different classes with the same names present in the Zoom PhysicsVectors package.

2.17 Reserved Member Names

Member names beginning with two underscores are reserved. The following reserved names are presently defined:

- `__base`N — DØOM implements base classes by treating them like additional member fields (this is the origin of the restriction against having data in virtual bases). These members are given names of the form `__base`N, where N is an integer. Members with names of this form should never be used in the input to `d0cint`.
- `__offset`N — If `d0cint` sees a member name of this form, it iterprets the member to be fixed at offset N within the C++ object. This can be useful in making external classes work with DØOM, where you want to "overlay" an external definition with one which works with DØOM (and has the same layout). See the `bitset` and `complex` classes for examples of this.
- `__packed` — Used to store any packed fields in the class. See Section 2.18.

2.18 Packed Fields

Sometimes, one wants to pack data tighter than it would naturally be stored. You can tell DØOM to attempt this with the `#pragma pack` directive. Note that how this is implemented depends on the specific I/O backend — at present, the DSPACK backend is the only one that handles packed data. (But it should be ok to specify `#pragma pack` directives even if you're using a backend that doesn't support them — they will just be ignored.) Also note that saving and restoring packed fields is likely to be much slower than ones that haven't been packed.

Only members of boolean or numeric type or collections of them may be packed.

The syntax of the packing directive is

`#pragma pack (`*packspec*`)` *member-names*

Here, *packspec* describes the sort of packing to do and *member-names* is a comma-separated list of the member names to be packed. They are looked up using the scope which is current at the point where the directive appears; the directive does not have to be lexically inside the class which it is modifying. A `#pragma transient` directive

should probably be placed inside a `#ifdef __D0CINT__` construction, to hide it from translators other than `d0cint`. (Or use `D0OM_TRANSIENT`; see section 9.2.)

The packing specification *packspec* consists of a comma-separated list of "*keyword=value*" pairs. The set of keywords accepted depends on the data type of the member, as listed below. All types, however, accept the `nbits` keyword, giving the requested number of bits to be used to store this member. This must be in the range 1–32.

- `bool` — Boolean fields have only the `nbits` keyword.
- Integer types — Besides the `nbits` keyword, integer types may also have the keywords `lo` and `hi`. These give the (inclusive) range of values allowed for this member. If either `lo` or `hi` is specified, both of them must be. If a range is specified, then the `nbits` keyword may be omitted. In that case, the number of bits will be chosen to be just large enough to hold the requested range.
- Floating point types — Besides the `nbits` keyword, the following keywords may be specified:
  - `scale` — If provided (and nonzero), the number being stored will be divided by this value before being stored. This allows one to scale numbers into the range allowed for fixed-point representations.
  - `signed` — If this keyword is set to "0", then a sign bit will not be stored: i.e., all numbers to be stored must be nonnegative. This defaults to "1".
  - `nmantissa` — The number of bits to use for the mantissa of the representation, excluding the sign bit (if any). All remaining bits left over from `nbits` after taking out the `nmantissa` bits and the optional sign bit are used for the exponent. If there are no more bits left (or if `nmantissa` wasn't specified), then a fixed-point representation is used. In the fixed-point case, the numbers being stored must be in the range $(-1, 1)$ (or $[0, 1)$ if no sign bit is being stored).

Here are some examples:

```
class Pack_Test
{
public:
  int a;
  int b;
  unsigned int c;
  bool d;
  float f;
  float g;

#ifdef __D0CINT__
# pragma pack (nbits=4) a;
```

```
# pragma pack (lo=10,hi=14) b;
# pragma pack (nbits=12,lo=0,hi=14) c;
# pragma pack (nbits=1) d;
# pragma pack (nbits=10,scale=20.5) f;
# pragma pack (nbits=10,scale=20.5,nmantissa=5) g;
#endif
};
```

A packing directive may also be used for arrays and collections of packable types:

```
class Pack_Test
{
public:
  int aa[3];
  unsigned int cc[3];
  bool dd[3];
  float ff[3];

  std::list<int> l;
  std::vector<int> v;

#ifdef __D0CINT__
# pragma pack (nbits=4) aa;
# pragma pack (nbits=12) cc;
# pragma pack (nbits=1) dd;
# pragma pack (nbits=10) ff;

# pragma pack (nbits=5) l;
# pragma pack (nbits=8) v;
#endif
};
```

2.18.1 DSPACK Implementation of Packed Fields

Here are some notes on the implementation of packing in the DSPACK backend.

The fields stored by DSPACK are always at least 32 bits wide. If a class has packed members, DØOM implements it on top of DSPACK by declaring a dummy field "`_packed`" to DSPACK and packing the data into there. DØOM maintains the metadata describing the format of the packed data itself, outside of DSPACK. (The information is stored as part of the comment string for the `_packed` field.) An array of N members is packed just like it was N separate members, and packed collections are stored like collections of integers.

This implies that a class instance must always take up an even multiple of 32 bits when stored, even if its contents could be packed to a smaller size.

In addition, the current implementation has a restriction that packed fields cannot cross a 32-bit boundary. This can cause DØOM to insert additional padding to achieve

this. This limitation may be removed in the future.

Packing directives may be freely changed (or added or removed) without affecting the ability to read old data.

3 DØOM I/O Interface

DØOM I/O proceeds though an abstract interface that is independent of the underlying I/O mechanism. The underlying I/O mechanism can be changed without requiring any modification of the I/O interface.

3.1 Class `d0Stream`

The main I/O interface class is an abstract class called `d0Stream`. The class `d0Stream` is intended to look and feel like a sequential or random access disk file, regardless of the underlying I/O implementation. The underlying I/O could in principle be to a single file, a sequential file list, federated files, a virtual stream, a database, shared memory, a network connection, or something else. Different concrete subclasses of `d0Stream` correspond to different underlying I/O mechanisms (see Fig. 7). In addition to concrete subclasses, class `d0Stream` has an abstract subclass `d0StreamDB`, which extends the `d0Stream` interface in ways that are appropriate for databases. Class `d0StreamDB` has its own concrete subclasses corresponding to different physical databases. Classes `d0Stream` and `d0StreamDB` are in the cvs module named `d0stream`.

3.1.1 Methods of `d0Stream`

The following methods are available in `d0Stream`.

`int close()`	Close a stream.
`d0_Ref_Any read(const d0Key* key=NULL, const std::string event_key="")`	Read an event.
`void write(const d0_Object& object, const std::string event_key="")`	Write an event.
`bool bind(const d0_Object& object, const d0Key& key)`	Associate a key with an object.
`void unbind(const d0Key& key)`	Delete the specified key.
`const d0Key* make_key(const char* key_string)`	Create a key and initialize it with a character string.
`d0_List<d0Key> keys() const`	Return a list of all known `d0Key`'s.
`d0_List<std::string> event_keys() const`	Return a list of all known event keys.

`int lookup(std::string` *event_key*`)`	Position the stream at the given event key.
`void seek(size_t` *offset*`)`	Seek to the specified offset.
`int tell()`	Return the current offset.

Note that there is no open method. Objects of type `d0Stream` are returned already opened when the are created.

3.1.2 Random Access and Keys

Methods of the `d0Stream` interface make use of two different kinds of keys. Methods, `read`, `bind`, and `unbind` take either a reference or a pointer to a `d0Key`. Methods `read`, `write`, and `lookup` take a string argument which is interpreted as an *event key*. Either kind of key can be used for keyed random access in certain situations. Both kinds of keyed access work similarly. A key is associated with an event, using either the `bind` method (for `d0Key`'s) or the event key argument of `write`. Keyed access is then accomplished using arguments of `read` (for either kind of key), or the `lookup` method (for event keys). Depending on the underlying I/O mechanism, the two kinds of keys may be implemented differently, or not al all. The following statements apply to the two kinds of keys.

1. For event oriented I/O mechanisms, event keys point to entire events and `d0Key`'s point to individual objects within an event.

2. Database I/O mechanisms implement both event keys and `d0Key`'s using a single underlying key mechanism.

3. Event keys are always represented as strings. `D0Key`'s are in principle polymorphic, although at present the only concrete implementation of class `d0Key` uses a single string its sole datum.

4. At most one event key can be associated with an object. With `d0Key`'s, many keys can be associated with the same object by calling `bind` repeatedly.

5. The DSPACK I/O mechanism (class `d0StreamDSPACK`) does not implement any random access mechanism that is accessible through the `d0Stream` interface.

6. The EVPACK I/O mechanism (class `d0StreamEVPACK`) implements keyed random access using event keys and via offset (`seek`/`tell`).

7. Both the DSPACK and EVPACK I/O mechanisms interpret the `d0Key` argument of method `read` as the name of a class within the current event (`bind`/`unbind` has no effect).

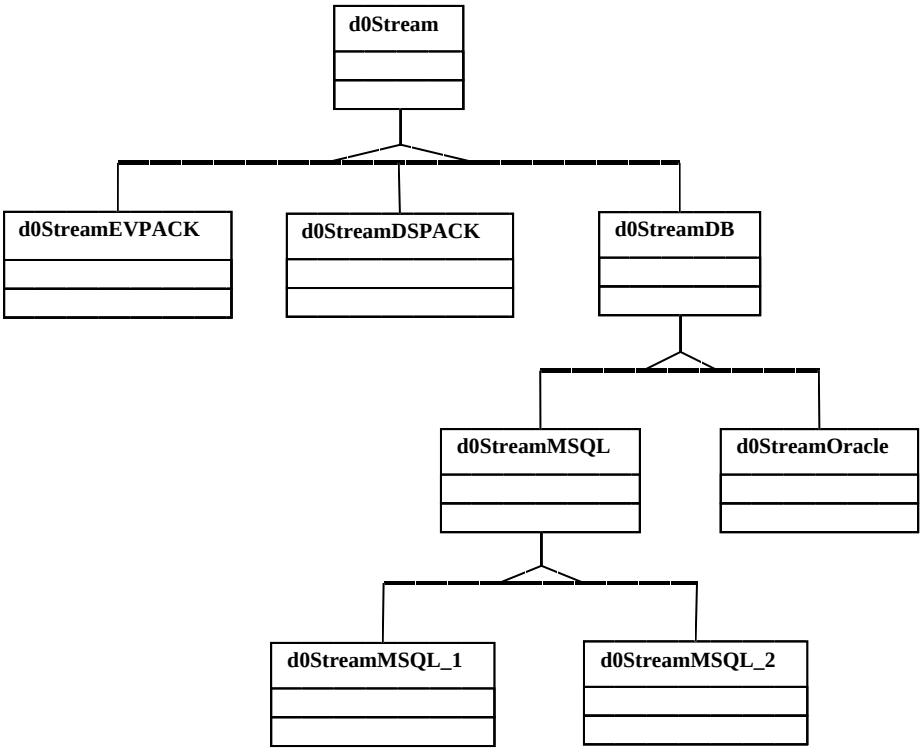

Figure 7: Inheritance diagram for `d0Stream`.

3.1.3 Random Access Recommendations

1. Use event keys in preference to `d0Key`'s unless you need a feature that is only available with `d0Key`'s (i.e. multiple keys per object or non-string keys). Event keys are implemented for more I/O mechanisms than `d0Key`'s, and their usage is more consistent between I/O mechanisms.

2. Use the EVPACK I/O mechanism for simple random access. Use a database only if you need database specific features, such as queries. EVPACK will give you better performance and better transportability.

3.1.4 Methods of `d0StreamDB`

The following methods are available in `d0streamDB`.

```
void open_transaction(bool readonly)   Open a transaction.
void commit_transaction()              Commit a transaction.
void abort_transaction()               Abort a transaction.
d0_List<d0_Ref_Any> query(const d0_String& classname,
                          const d0_String& condition,
                          const d0_String& tables="",
                          const d0_String& fields="")
                                       Query database.
```

The above methods are designed to support transaction and query operations with a real underlying database, such as Oracle.

The `query` method has the following four arguments, of which the last two are optional.

1. *Classname.* This is the C++ name of a concrete C++ class. The returned `d0_Ref_Any` pointers can be downcast (using `D0_REFCAST`) to the specified class, or a base class of the specified class, but not to a class that is derived from the specified class.

2. *Condition.* This is in general an arbitrary SQL select statement clause, such a `where` or `order by` clause.

3. *Tables.* This argument contains a comma separated list of tables (not including the table corresponding to the *classname* argument) to add to the `from` clause of the generated select statement. A non-null *tables* argument would typically be required for queries containing join operations, or in general any time table-qualified fields appear in the *condition* argument.

4. *Fields.* This argument contains a comma separated list of additional fields to fetch in this query. Fields can be specified in the forms "*column*" for single-table queries, or "*table.column*" for multitable queries. The *fields* and *tables* arguments have no effect on the returned list of pointers, but only on whether the query generates an error or not.

Name Mangling Issues in Queries. In general, the *condition*, *tables*, and *fields* arguments contain raw SQL that is edited directly into an SQL `select` statement. Therefore, table and column names appearing in these arguments should correspond to database table and columns names, which are not in general the same as the corresponding C++ class and field names.

All three of the SQL arguments allow C++ names to be used instead of database names by enclosing them in `$(...)`. The string `$(`*class.field*`)` appearing in any of these arguments is converted into a string of the form *table.column*. The parsing of `$(...)` expressions is governed by the following rules.

1. Simple scalar fields of C++ classes are specified in the form `$(`*class.field*`)`.

2. Array fields are specified using normal C++ notation: `$(`*class.field*`[`*n*`])`. For this to work, the database schema must be such that the array is not broken out as a separate table. In general, this will be true for "short" arrays.

3. Fields of contained classes are specified in the form `$(`*class.subclass.field*`)`. In this case, *class* is translated into the database table, and *subclass.field* is translated into the database column. For this to work, the database schema must be such that contained class is not broken out as a separate table. This should usually be true.

4. Template class names, such as containers, are allowed. They are specified in normal C++ notation.

5. Base classes are treated like contained classes. Field names for base classes are `__base1`, `__base2`, etc., where the number of the base class corresponds to their order in the class definition. A typical C++ field expression would be `$(`*class*`.__base1.`*field*`)`.

6. The *class* can be omitted in single table queries. The following forms are permitted: `$(`*field*`)`, `$(.`*field*`)`, `$(.`*subclass.field*`)`. The form `$(`*subclass.field*`)` (without a leading period) is not allowed, because in this case *subclass* would be interpreted as a class name.

7. Mixed forms, where the class is specified in C++ format, but the field is specified in database format are allowed, for example `$(`*class*`.)objid`. This syntax is useful to specify database columns that do not correspond to any C++ class field (i.e. metafields).

8. Some C++ class fields, such as containers and large arrays are broken out as separate database tables. The `$(...)` notation does not provide a way to reach into such auxilliary tables. It may still be possible to reach into such tables by using a join operation which matches the container field of the parent class with the object id field of the container class.

3.2 Factory Class d0StreamFactory

A factory class d0StreamFactory exists to provide users with instances of subclasses of d0Stream. The factory class serves to shield end users from having to directly instantiate d0Stream's concrete subclasses. The factory class also reduces the amount of compile-time and link-time coupling between user code and the d0Stream subclasses. In particular, d0Stream does not have the D0OPEN disease of having link time coupling to every I/O method in the library. Users must explicitly specify linking of d0Stream subclasses that they want in a program.

The class d0StreamFactory is a singleton [5]. It has two methods of interest to users:

```
static d0StreamFactory* locateStreamFactory ();
d0Stream* make_d0Stream (const d0StreamName& name,
                         const char* streamType = "",
                         int mode = ios::in,
                         const string& = "");
```

The method locateStreamFactory returns a pointer to the factory singleton. The method make_d0Stream creates an instance of an open d0Stream subclass. The argument streamType is a character string that specifies which type of subclass to create (use "DSPACK" for d0StreamDSPACK, "EVPACK" for d0StreamEVPACK, "ORACLE" for d0StreamORACLE_1, or "CORBA" for d0StreamCORBA_1). The argument streamType may be empty or null when opening files for reading. In that case, d0StreamFactory tries to figure out which type of d0Stream subclass to instantiate. The argument "name" specifies the name of a physical stream (e.g. a filename). The name can be specified as an ordinary C++ double-quoted string. The argument "mode" determines how the specified d0Stream is to be opened. Mode can take the values ios::in (read), ios::out (write), or ios::app (append) defined in the standard header iostream. The final string argument is not interpreted by d0StreamFactory, but is passed to the create static method of the concrete d0Stream subclass. It typically consists of a sequence of *name=value* pairs. The interpretation of this string is up to the I/O mechanism. Presently, this is only used by EVPACK; it recognizes a compression_level= argument.

Note that d0StreamFactory::make_d0Stream will return null if the code to implement the requested stream type has not been linked into the program. Here is an example of the use of d0StreamFactory and d0Stream.

```
// Get pointer to factory.

d0StreamFactory* factory = d0StreamFactory::locateStreamFactory();

// Create an instance of d0Stream.

d0Stream* dsin = factory->make_d0Stream("myfile.ds",
                                       "DSPACK", ios::in);
```

```
if (!dsin) {
  cerr << "Can't open input file\n";
  exit (1);
}

// Read event.

d0_Ref<Event> revent = D0_REFCAST((Event))(dsin->read("Event"));
```

3.3 Output Filters

When writing, you can supply an output filter to control which objects get output. Such a filter is an instance of a class which derives from `d0_Output_Filter_Base`. See the header file for the definition of the interface; essentially, it provides a method which takes a pointer to an object as an argument and returns a flag saying whether or not that object should be written.

When an object is vetoed by this mechanism, pointers from it are not followed. Thus, vetoing an object implicitly vetoes all other objects which can only be reached via that object. References to vetoed objects are set to null.

To install an output filter, call `d0Stream::set_output_filter`, passing to it a pointer to an output filter instance. This object should have been allocated off the heap with `new`. The stream will take ownership of the filter; it will get deleted automatically along with the stream (or the next time the output filter is changed). To remove the output filter, call `set_output_filter` with a null pointer. The current output filter may be retrieved with `output_filter`.

There is one concrete output filter implementation available in the library, named `d0_Output_Filter`. It has the ability to either select or veto individual objects and also to select or veto all objects of a given class. See the header file for more information.

3.4 Memory buffers

EVPACK format data may also be read from and written to memory buffers. This is useful for applications which want to send event data over the network.

3.4.1 Output

Output to memory buffers is done using the class `d0StreamEVPACK_Buffered_Output` (in `d0om_ds`). Here is its class definition:

```
class d0StreamEVPACK_Buffered_Output
  : public d0StreamDSPACK_Base
//
// Purpose: Evpack stream for writing using
```

```
//          an arbitrary hook function.
//
{
public:
  // Constructor, destructor.
  // OPTARG is additional arguments to pass to evpack.
  d0StreamEVPACK_Buffered_Output (const std::string& optarg = "");
  ~d0StreamEVPACK_Buffered_Output () {}

  // Write object.
  virtual void write (const d0_Object &object,
                      const std::string& event_key);

  // Supply this in a derived class.
  // It will be called for every evpack record written.
  virtual std::streamsize writebuf (const char* data,
                                    std::streamsize n) = 0;
};
```

This class supports the `d0Stream` interface. However, it cannot be constructed through the `d0Stream` factory; you must explicitly create the objects. The usual `write` method will write out a tree of objects.

For each EVPACK record generated, the `writebuf` method will be called. This is a pure virtual function in `d0StreamEVPACK_Buffered_Output`, so you should derive from that class and supply a definition for this method. The data will be in the buffer described by the pointer `data` and the size `n`. Note that the buffer will not remain valid after `writebuf` returns, so you should arrange to copy the data somehow.

An example of the use of this class is in one of the `d0om_ds` test programs, `d0om_ds/tests/evpack/twrite_evpack_buf.cpp`.

3.4.2 Input

Input to memory buffers is done using the class `d0StreamEVPACK_Buffered_Input` (in `d0om_ds`). Here is its class definition:

```
class d0StreamEVPACK_Buffered_Input
 : public d0StreamDSPACK_Base
//
// Purpose: Evpack stream reading from a memory buffer.
//
{
public:
  // Constructor, destructor.
  // OPTARG is additional arguments to pass to evpack.
  d0StreamEVPACK_Buffered_Input (const std::string& optarg = "");
```

```
  ~d0StreamEVPACK_Buffered_Input () {}

  // Set the buffer for further input to the chunk of
  // memory described by DATA and LENGTH.
  // Use (0, 0) to clear the buffer.
  // If an attempt is made to read past the end of the buffer,
  // evpack will behave as if EOF was hit.
  void set_buffer (void* data, int length);
};
```

This class supports the `d0Stream` interface, including the usual `read` method. However, it cannot be constructed through the `d0Stream` factory; ou must explicitly create the objects. In addition, random access will not work.

Instead of reading from a file, this stream class reads from a memory buffer, which is supplied via the `set_buffer` method. The arguments to `set_buffer` are the address of the buffer and its length. If the end of the buffer is reached, attempts to read from the stream will behave is if EOF was hit.

The method `set_buffer` may be called as may times as desired, in order to declare a new buffer to a stream. The old buffer is forgotten. To clear the stream's pointer to the buffer (so that the buffer may be deleted), do `set_buffer (0, 0)`.

An example of the use of this class is in one of the `d0om_ds` test programs, `d0om_ds/tests/evpack/tread_evpack_5.cpp`.

3.4.3 Embedded Dictionary Records

The usual organization of a DSPACK file consists of a dictionary record at the start of a file, then some number of data records, and a final dictionary record at the end of the file. In order to be able to interpret the data records, the proper dictionary record must have been read first.

This organization works fine when data is being read from a file. However, it is inconvenient if events are being sent over the network, especially if they are being accumulated from several sources (as in the case of events being received from level 3). Therefore, EVPACK has an option to embed the dspack dictionary information within each EVPACK record. (This will, of course, increase the size of the data being written.) To enable this, add the string "`embed_defs`" to the `optarg` parameter used when creating an EVPACK output stream.

On input, embedded DSPACK dictionary records are handled automatically. Because it is relatively expensive to process a dictionary record, such a record is read only if it appears to be different from the previously read dictionary record. (This determination is made by using a MD5 checksum of the dictionary record.)

4 Using DØOM

A DØOM program should do the following things.

1. System initialization.

   ```
   call d0om_init("test");
   ```

   (Declared in `d0om/d0om_init.hpp`.) The main purpose of this call is to initialize the DØOM type system. The interpretation of the single character string argument is I/O mechanism specific. In the case of DSPACK programs, the argument is passed to the DSPACK initialization routine `dsinit`, which uses it to identify the client to the server.

2. Get an instance of class `d0Stream` using the stream factory class (see sec. 3.2).

   ```
   d0StreamFactory* factory =
      d0StreamFactory::locateStreamFactory();
   d0Stream* dsin = factory->make_d0Stream("myfile.ds", "DSPACK",
                                          ios::in);
   if (!dsin) {
     cerr << "Can't open input file\n";
     exit (1);
   }
   ```

   The instance of `d0Stream` is returned already opened.

3. Call I/O methods of `d0Stream`, such as `read`, `write` and `bind` (see sec. 3.1).

4. Call method `close` and delete the instance of `d0Stream`.

4.1 How to Make a Class Persistent

In general, a programmer must do three things to make a class persistent.

The first step is to make the header file for the class conform with the persistent object model outlined in section 1.1. This means inheriting from `d0_Object` or another persistent class, and ensuring that persistent data fields are one of the allowed types. The class must also have a default constructor (or a `d0_Input_Info` constructor, see Sec. 2.10).

The second step is to include the following macro in the body of the class declaration:

`D0_OBJECT_SETUP(`*myclass*`);`

The third step in making a persistent class is to run the DØOM preprocessor `d0cint`. The command to run `d0cint` is as follows:

```
d0cint myclass_lnk.cc -Iinclude myclass.hpp
```

In the above example, `myclass.hpp` is the input header file and `myclass_lnk.cc` is an output file. The programmer should compile and link the file `myclass_lnk.cc` into his program. The include path for the `d0cint` command should be the same as would be used in a normal compilation. Additional `-I` or `-D` switches may be specified as needed. (A makefile scrap is provided to perform this step for you when you build a library with SRT. See Section 5.8.)

A persistent class can contain both persistent and transient fields. To make a field transient, use the `#pragma transient` directive (see Section 2.7). A data element that is not one of the allowed persistent data types is transient by default. In the latter case, `d0cint` will issue a warning message.

When `d0cint` reads a header, it defines the preprocessor symbols '`__CINT__`' and '`__D0CINT__`' in order to identify itself. Thus, if a header contains constructs which `d0cint` is not understanding, they can be enclosed in an '`#ifndef __CINT__`' or '`#ifndef __D0CINT__`' to keep `d0cint` from trying to interpret them. (The practical distinction between these symbols is that '`__CINT__`' is also defined by `rootcint`, while '`__D0CINT__`' is not.

4.2 Reference Headers

One must link into an application the linkage (`_lnk`) object for every class which is to be used. Since there are normally no external references to the linkage object, this can be problematic if the object gets placed in a library.

Reference headers offer one solution to this. In addition, they provide a mechanism of ensuring that all modules of the application were compiled with compatible definitions for class layouts.

The idea is this. For each class which `d0cint` defines in a `_lnk` file, it also creates a global symbol definition of the form `d0omhash_`*classname*`_`*hashcode*, where *hashcode* is a 8-character string which depends on the definition of the persistent fields of the class. If you supply the switch '`-ref ref_h`' to `d0cint`, it will write to `ref_h` a C++ header containing external references to those global symbols. This reference header can then be included by the header defining the class. (It should probably be enclosed in a '`#ifndef __CINT__`' to prevent `d0cint` from trying to read it before it is created.) This should ensure that the linkage object is loaded from a library when it is needed. In addition, if you try to link together objects which were compiled with different versions of the class, you'll get undefined symbols at link-time.

As of `v00-17-00`, DØOM can automatically make some of these references. For a given class `A` defined in `A.hpp`, its linkage file will contain references to the linkage files for all classes which `A` depends on. Thus, if you have a class which is used only by reference from another class, you don't need to include its reference header.

4.3 `d0cint` Command Reference

This section summarizes the syntax for the `d0cint` command. Note that in most cases, it should not be necessary to run `d0cint` directly. It should, instead, be run through `d0om_linkage.mk` (see Section 5.8).

Command format:

- `d0cint` *output-file options input-headers*

The input to `d0cint` is one or more C++ headers, named in *input-headers*. The output is a C++ source file, named by *output-file*.

If any argument starts with '`@`', the remainder of the argument should be a file name. The contents of that file are then inserted into the argument list.

The recognized options are listed below. (All options recognized by `cint` are recognized by `d0cint`, but only those believed to be useful for `d0cint` are listed below.)

`-I`*directory*	Add an additional directory to the include path. Directories are searched in the order in which they are specified. For each include directory *directory*, the additional directories *directory*`/CINT/d0cintinc` and *directory*`/d0om_zm/d0cintinc` are also added to at the front of the include path (provided that they exist).
`-D`*macro*	Define a macro for the preprocessor. Only macro definitions without a parameter are likely to work well. (I.e., use `-DFOO`, not `-DFOO=bar`.)
`-t`	Trace files included by `cint`.
`-T`	Echo the input read by `cint` to standard output, along with additional internal `cint` debugging information. This can be useful for localizing parsing problems with template or macro definitions. (Note that this is raw output from `cint`'s reader, and may contain some artifacts due to the details of how `cint` parses its input. In particular, `cint` sometimes backs up in the input stream; this can cause some text to be echoed twice.)
`-v`	Dump out the arguments internally passed to the `cint` main entry point.
`-q`	Suppress the messages listing which classes `d0cint` generated linkage information for.
`-ref` *filename*	Write a reference header to *filename*. See Section 4.2 for more information.
`-dep` *filename*	Write dependency information to *filename*. This contains the depdencies which the linkage source file has on the headers.

4.4 Predefined Macros

When `d0cint` runs, it defines the following macros to be the values of the corresponding environment variables (without the leading underscores):

- `__SRT_ARCH`
- `__SRT_CXX`

- `__SRT_QUAL`
- `__BFARCH`

The macros `__D0CINT__` and `__CINT__` are also defined.

4.5 Environment Variables For Debugging

There are several environment variables which DØOM examines to control whether to make debugging dumps.

- `D0OM_DUMP_DICTIONARY`: If this environment variable is defined, DØOM will dump (to stdout) a description of the types that it knows about at the end of the initialization stage.
- `D0OM_DS_DUMP_MAPTABLE`: If this environment variable is defined, DØOM will dump (to stdout) the internal details of the mappings it builds between DSPACK and C++.

4.6 Functions for Debugging

The following two functions may be useful for tracking down memory leaks involving objects read from DSPACK. Both of these functions are defined in the header `d0om_ds/debug.hpp`.

- `d0om_DS::list_objects (std::ostream& os, bool mask_ptr = false)` — Dump to `os` a list of all dspack objects which still have live references. The information printed for each object includes the address (if it has been created), type name, file offset, and dataset index. If `mask_ptr` is true, then the pointer and file offset fields will be suppressed in the output (this is for regression testing).
- `d0om_DS::summarize_objects (std::ostream& os)` — This is similar, but it prints only one line per object type, giving the number of objects with live references of each type.

5 How to Compile and Link a DØOM Program

This section contains some practical instructions on how to get access to DØOM.

5.1 Requirements

The examples in this section assume that the following products are available locally: ups, SoftRelTools, and d0cvs. If d0cvs is unavailable, vanilla cvs should work in most cases. The only d0cvs specific command used in this section is "lscvs." The examples shown here are known to work on d0chb. The same procedures should work

at remote unix computers. It is not necessary to be local to the cvs repository to use cvs commands.

DØOM is presently known to work with KCC 3.4g, gcc 2.95, and SGI CC 7.3 on Irix 6.5, KCC 3.4g, and gcc 2.95 on Linux, and KCC 3.4g on Digital Unix 4.0d. Most packages should also work on NT with the Microsoft compiler.

5.2 Setups

In general, before compiling or linking a program that uses d0library code, including DØOM, several setups may be necessary. The most basic setup, which gives access to released code, and to the SoftRelTools utilities is:

```
setup D0RunII test
```

This setup defines the environment variables `BFROOT`, `BFCURRENT`, `BFDIST`, and `BFARCH`. To gain access to the cvs repository, use the following setup command:

```
setup d0cvs
```

This setup defines the environment variables `CVS_DIR`, `DOCVS_DIR`, and `CVS_ROOT`. It should also set up the C++ compilers. (If you want to change the C++ compiler you're using, you'll need to change the definition of `BFARCH`.) For more information about the D0 cvs and SoftRelTools environments, refer to the D0 code management web page [4].

5.3 CVS Libraries

Use the d0cvs command "`lscvs`" to see a list of currently defined cvs modules. The cvs modules relevant to the DØOM system are shown in Table 7. Package `DSPACK` contains the distribution from CERN, slightly modified to build under SRT. `CINT` contains the cint C++ interpreter from HP Japan, with numerous local modifications. `d0_util` contains a number of utility classes used by DØOM. `d0om` is the core portion of DØOM, and `d0stream` is the abstract part of the stream code. `d0om_ds` is the DØOM-DSPACK interface. The Oracle and Corba interfaces are in `d0omORACLE` and `d0omCORBA`, respectively.

5.4 DØOM Source Code

Already released DØOM source code can be found in the SoftRelTools packages area. The root directory for DØOM source code is

```
$BFDIST/packages/d0om/<package-version>.
```

Table 7: DØOM-related cvs modules.

CVS module	Libraries	Description
d0omORACLE	-ld0omORACLE	DØOM Oracle interface.
d0omCORBA	-ld0omCORBA	DØOM Corba interface.
d0om_zm	-ld0om_zm	Adapters for Zoom classes.
d0om_ds	-ld0om_ds, -lstream_ds	DØOM DSPACK interface.
d0stream	-lstream	Stream interface code.
evpack	-evpack	EVPACK package.
d0om	-ldoom	Basic DØOM code.
DSPACK	-ldspack	DSPACK package.
CINT	-lcint	CINT C++ interpreter.
d0_util	-ld0_util	Utility classes.

5.5 Releases

Already compiled DØOM C++ libraries and the `d0cint` executable can be found in the SoftRelTools releases area. Some significant directories in the releases area are as follows:

```
$BFDIST/releases/<release-version>              Release root
$BFDIST/releases/<release-version>/bin/$BFARCH  Executables
$BFDIST/releases/<release-version>/lib/$BFARCH  Link libraries
$BFDIST/releases/<release-version>/include      Include path
$BFDIST/releases/<release-version>/include/d0om D0OM include files
$BFDIST/releases/<release-version>/d0om         Link to D0OM package
$BFDIST/releases/<release-version>/DSPACK       Link to DSPACK package
```

The release version can be a tag, such as "`current`" or "`test`", or a version number. The include path for compilers and `d0cint` should be specified as `-I$BFDIST/releases/<`*release-version*`>/include`. The link path for link libraries should be specified as `-L$BFDIST/releases/<`*release-version*`>/lib/$BFARCH`. The libraries needed for linking are listed in Table 7. In addition to these, the zoom libraries `-lExceptions` and `-lZMutility` will be needed; and if you use EVPACK, `-lz` for zlib. You also need to link against the system's fortran library. This varies from system to system; here are additional link flags for various systems:

Irix	`-lftn`
AIX	`-lxlf90 -lxlf`
Digital Unix	`-taso -lUfor -lfor -lFutil -lots -lm -lc_r`
Linux	`-lf2c -ldl -lcrypt`

(Note that `-taso` is required for any application which links with DSPACK on Digital Unix.)

If you are using the `d0Stream` interface with DSPACK, you'll need to be sure that the module `d0StreamDSPACK.o` gets loaded from the library `libstream_ds.a`. There are two general ways of doing this:

- Extract the module from the library and include it explicitly in your link command.

- Include the header `d0StreamDSPACK.hpp` explicitly in some other module which you are linking in. Including this header creates a reference to the `d0StreamDSPACK` module, causing it to be loaded from the library.

Some makefile fragments are now available to try to simplify linking with these libraries.

Including `d0om/arch_spec_d0om.mk`, for example, will define the make symbol `D0OM_LIBES`, containing the library specifications which you should add to your link command in order to link with DØOM. If you want to link everything with DØOM, then you could add the value of `D0OM_LIBES` to `LOADLIBES`. (This is not done by default, in case you only want to link some programs with the package.)

Here is a list of these scraps:

- `DSPACK/arch_spec_dspack.mk`: Defines `DSPACK_LIBES`. (Note: On OSF, this will also add `-taso` to `LDFLAGS`, which is required for linking with DSPACK on that platform.)

- `d0om/arch_spec_d0om.mk`: Defines `D0OM_LIBES`. This links with only the 'core' DØOM library. It does not link with the stream code, or with any of the I/O backends.

- `d0stream/arch_spec_d0stream.mk`: Defines `D0STREAM_LIBES`. This links with d0om and the stream code. It does not link with any of the I/O backends.

- `d0om_ds/arch_spec_d0om_ds.mk`: Defines `D0OM_DS_LIBES`. This links with DØOM, the stream code, and the DSPACK I/O backend. This scrap also defines `D0OM_EV_LIBES`, which includes evpack in the link.

If you are using Zoom classes in persistent classes, you should also link with `-ld0om_zm`.

5.6 Becoming a Developer

It is possible to create a private releases area. When you do this, you must do a full recompilation of any packages that you are interested in, either from the packages area, or of code extracted from cvs. You might do this because you want a more recent version of the code than can be found in the releases area, or because you want to be able to modify the library code. The following is an example session that shows one does this:

```
setup D0RunII
setup d0cvs
mkdir ~/myrelease
cd ~/myrelease
```

Table 8: DØOM header files.

DØOM class category	Header file
Base class	d0om/d0_Object.hpp
Atomic typedefs	d0om/d0_basic_types.hpp
Strings	d0_util/d0_String.hpp
Container classes	d0_util/d0_Vector.hpp
	d0_util/d0_List.hpp
	d0_util/d0_PVector.hpp
	d0om/d0_RVector.hpp
	d0_util/d0_PList.hpp
	d0om/d0_RList.hpp
Iterators	d0_util/d0_Iterator.hpp
	d0_util/d0_PIterator.hpp
	d0om/d0_RIterator.hpp
References	d0om/d0_Ref.hpp
Stream interface	d0stream/d0Stream.hpp
	d0stream/d0StreamFactory.hpp
	d0stream/d0Key.hpp

```
newrel -t current test   # Creates directory ~/myrelease/test
cd test
addpkg -h d0om_ds        # Fetch D0OM DSPACK interface code from cvs.
addpkg -h d0stream       # Fetch D0OM stream code from cvs.
addpkg -h d0om           # Fetch D0OM code from cvs.
addpkg -h d0_util        # Fetch utility classes from cvs.
addpkg -h CINT           # Fetch CINT code from cvs.
addpkg -h DSPACK         # Fetch DSPACK code from cvs.
gmake installdirs        # Make some directories.
gmake                    # Compile everything
```

Refer to the code management web page [4] for more information about developer tools.

5.7 DØOM Header Files

Table 8 contains a list of DØOM header files. DØOM header files are included in user programs with the following type of `#include` statement:

```
#include "d0om/d0_Ref.hpp"
```

Note the inclusion of the directory (such as "`d0om/`") in the include statement.

5.8 Generating Linkage Files

DØOM supplies a makefile scrap which can be used to generate the linkage files. To use this, include the line

```
include d0om/d0om_linkage.mk
```

in your `GNUmakefile` before `standard.mk`. Before including it, you should define the variable `LINKAGE_HPP` to be a list of headers to process through `d0cint` (the file name only, without any leading path component) and `LINKAGE_HPP_DIRS` to be a list of directories in which to search for those headers. The standard include path will be searched, if necessary, to find these directories. If you don't specify `LINKAGE_HPP_DIRS`, it will default to the name of the current package. If `LINKAGE_REF` is defined, a single object file incorporating all reference headers will be created in `$(libdir)`. The object file name is `$(LINKAGE_REF)_ref.o`.

By default, both the C++ sources and the reference headers will be generated in `$(workdir)`. These defaults can be changed by setting the variables `LINKAGE_CPP_DIR` and `LINKAGE_REF_DIR`. The C++ sources will be added to `LIBCPPFILES`, and (provided `LINKAGE_CPP_DIR` wasn't overridden) appropriate vpath directives will be issued to allow these sources to be found.

If you wish to prevent `d0cint` from running but want to retain all the dependencies which `d0om_linkage.mk` sets up, define the variable `DONT_RUNCINT`.

By default, when `d0cint` is run from `d0om_linkage.mk`, it will produce no output if it is successful. But if the make variable `VERBOSE` is set, `d0cint` will echo a list of the classes for which it produced linkage information. Further, if you expect `d0cint` to produce warnings for some of your files, define the make variable `D0OM_LINKAGE_ERRORS_EXPECTED`. This will cause `d0om_linkage.mk` to prefix all `d0cint` output with '`-->`', which will prevent the release procedures from claiming that your package is broken.

Example:

```
LIB := libfoo.a

LIBCPPFILES := $(wildcard *.cpp)

# Process the headers foo/class1.hpp
#                 and foo/class2.hpp.
LINKAGE_HPP_DIRS := foo
LINKAGE_HPP := class1.hpp class2.hpp

include d0om/d0om_linkage.mk
include SoftRelTools/standard.mk
```

5.9 Controlling Generation of Linkage Information

When `d0cint` is run on a header file, it generates dictionary information only for the classes which are defined in that header. In particular, if `A.hpp` defines class `A`,

`B.hpp` defines class `B`, and `B.hpp` includes `A.hpp`, then running `d0cint` on `B.hpp` will not generate dictionary information for class `A` — you must run `d0cint` separately on `A.hpp` and `B.hpp`, even if class `B` uses class `A`.

In some cases, however, more control over this process is desirable. This can be had with the `#pragma linkage` and `#pragma nolinkage` directives. A directive of the form

`#pragma linkage` *class-names*

tells `d0cint` to generate dictionary information for *class-names* (which should be a comma-separated list). On the other hand,

`#pragma nolinkage` *class-names*

tells it not to generate dictionary information for classes for which it would ordinarily do so. Note that these directives take effect only if they are in the outermost header file (i.e., the one which you named to `d0cint`).

These directives should usually be used inside an `#ifdef __D0CINT__` construction in order to hide them from translators other than `d0cint`. (Or use `D0OM_LINKAGE` and `D0OM_NOLINKAGE`; see section 9.2.)

One use of this facility is to tell `d0cint` to generate dictionary information for particular template instantiations (see Section 2.14). It may also be useful for using classes from external libraries.

One additional mechanism is available. If a class declares a typedef name called `d0om_autolink` (it doesn't matter what it is defined as), then that class will always have linkage information generated, provided that it is used, directly or indirectly, by another class for which linkage information is being generated. One application of this is to get linkage information generated for template classes, without requiring the user to explicitly declare all instantiations of the class. (`#pragma linkage` won't work in this case, because it can only be applied to a specific instantiation, not a template.) This mechanism is used, for example, to ensure that if user uses a `map<K, T>` template, linkage information is generated for the associated `pair<const K, T>` class.

As an example, here is the definition of `std::pair` as used by `d0cint`:

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;
  pair();
  pair(const T1& x, const T2& y);

  // Flag that this class should have full linkage information
  // generated in each header where it's used.
  typedef int d0om_autolink;
};
```

Note that this mechanism should not be used for concrete classes deriving from `d0_Object`; otherwise, you'll probably end up with multiply defined symbols at link time.

6 Utility Programs

This section lists various utility programs which are available.

6.1 dsdump

The program `dsdump` can be used to dump out the contents of a DØOM DSPACK (or EVPACK) file. It takes a single argument, the name of the input file, and dumps out all records in the file.

Here is an example of the output from the program. Note that object instances are identified by the class name followed by an integer ID number.

```
Read record 1 (d0om_ds format version 5)
d0om_DS_Head:1
  d0_Ref<d0_Object> head: T3:2

T3:2
  T3::t3foo bar:
    d0_Int a: 98
    d0_Int b: 99
  d0_Ref<T2<int> > r1: T2<int>:3
  d0_Ref<T2<float> > r2: T2<float>:4

T2<int>:3
  d0om_Unknown_List<T1<int> > b:
      d0_Int a: 4
      d0_Int a: 7
      d0_Int a: 10

T2<float>:4
  d0om_Unknown_List<T1<float> > b:
      d0_Float a: 9.4
      d0_Float a: 6.2
      d0_Float a: 3.4
```

Note that the data format must be version 5 (written by `v00-15-00`) or later for this to work.

With the argument `--sizes`, `dsdump` will also dump out the sizes of all nonempty DSPACK datasets in the file. The argument `--notree` will suppress the usual dump of the event contents.

With the option `--raw`, `dsdump` will print collections of numeric types as raw hexadecimal dumps. This is useful for looking at raw data. If, in addition, the `--byteswap` option is given, the values being printed in the hex dump will be byteswapped on little-endian machines.

With the option `--examine`, `dsdump` will invoke the DSPACK debugger for each event read. (See Section 6.2.)

The option `--precision=`*prec* may also be specified, to set the precision used when writing floating point numbers.

The option `--versions` will cause `dsdump` to print the stored version numbers for all classes.

The option `--key=`*key* will cause `dsdump` to seek to the event with event key *key* before starting to dump. (This works only for EVPACK format files.)

The option `--count=`N will cause `dsdump` to exit after printing N events.

The option `--only=`*pattern* will cause `dsdump` to print only objects with names matching the regular expression *pattern*. Any commas in *pattern* are replaced with '`|`', so that a comma-separated list may be used. This option is implemented only on unix platforms.

6.2 DSPACK debugger

There is also a DSPACK "debugger," which can be used to examine the raw DSPACK structures. There are several ways to start the debugger. The first is by running `dsdump` with the `--examine` switch (see Section 6.1). In this case, the debugger will be called after each event is read. If you are reading files in EVPACK format (as opposed to plain DSPACK format), this is the only way to use the debugger to look at the data.

The second way to start the debugger is with the standalone `dstest` program.

The third way is to run DSPACK in client-server mode. The first step is to start the server with '`dsserver`':

```
$ dsserver
Starting server snyder_d0linux01
```

Next, start the debugger with '`ds db`':

```
$ ds db
DSPACK 1.520, 10 Mar 1998    (dsdb, server: snyder_d0linux01)
dsdb>
```

Once the debugger has been started by any of these methods, saying '`help`' will give you a list of valid commands.

You can open an input file with '`input`'; '`read`' will then read the next event from the file. (But if you started the debugger through `dsdump --examine`, an event should already be loaded.)

```
dsdb> input tmp/Linux2-KCC_3_3/d0om_ds/ttmpl.ds
```

```
Read definitions
dsdb> read
Read one event
```

You can see what DSPACK data sets are defined with '`ls -l`':

```
dsdb> ls -l
Name              Type  Id  Sect   Size    Items   Nent   Bytes   Free
d0om_DS_Evdat       S   67    1      3       3       1      12      0
Object              S   68    1      1       1       0       0      0
Ref_d0_Object_      S   69    1      1       1       0       0      0
d0om_DS_Head        S   70    1      1       1       1       4      0
T3__t3foo           S   71    1      2       2       0       0      0
T1_int_             S   72    1      1       1       3      12      0
_LT1_int_           S   73    1      2       2       0       0      0
T2_int_             S   74    1      2       1       1       8      0
T1_float_           S   75    1      1       1       3      12      0
_LT1_float_         S   76    1      2       2       0       0      0
T2_float_           S   77    1      2       1       1       8      0
Ref_T2_int__        S   78    1      1       1       0       0      0
Ref_T2_float__      S   79    1      1       1       0       0      0
T3                  S   80    1      4       3       1      16      0
Int                 S   81    1      1       1       0       0      0
UInt                S   82    1      1       1       0       0      0
```

The '`Nent`' column gives the number of entries in that data set in the current record.

To print the definition of a data set, enter its name:

```
dsdb> T3
typedef struct T3__t3foo DS_VARIABLE {    /*T3::t3foo;s */
  int_t a;                                /*a */
  int_t b;                                /*b */
} T3__t3foo;

typedef struct Ref_T2_int__ DS_VARIABLE {    /*d0_Ref<T2<int> >;r */
  void *ptr;                                 /*T2_int_ */
} Ref_T2_int__;

typedef struct Ref_T2_float__ DS_VARIABLE {  /*d0_Ref<T2<float> >;r */
  void *ptr;                                 /*T2_float_ */
} Ref_T2_float__;

typedef struct T3 DS_VARIABLE {    /*T3;o */
  struct T3__t3foo bar;            /*bar */
  struct Ref_T2_int__ r1;          /*r1 */
  struct Ref_T2_float__ r2;        /*r2 */
```

```
} T3;
```

This also prints the definitions of any contained data sets. Note that the names of the data sets may not match the names of the C++ classes — they are sometimes "mangled" in order to conform to the restrictions which DSPACK puts on names. You can see the full name, though, in the comment field on the first line of the definition.

To dump out the data in a data set, enter the data set name preceded by an asterisk:

```
dsdb> *T3
 Data set: /T3
 Type:                  Structure   (at: 0x5000669c)
   1  bar               T3__t3foo
        Type:                 Structure   (at: 0x5000669c)
          1  a                Integer        : 98
          2  b                Integer        : 99
   2  r1                Ref_T2_int__
        Type:                 Structure   (at: 0x500066a4)
          1  *ptr             Pointer        : 0x5000656c T2_int_(1)
   3  r2                Ref_T2_float__
        Type:                 Structure   (at: 0x500066a8)
          1  *ptr             Pointer        : 0x50006624 T2_float_(1)
```

This will dump out all entries in the data set. To dump only some of the entries, you can use a Fortran-like array notation, like '`T3(3)`' or '`T3(2:5)`'.

To exit from the debugger, use '`quit`'. If you started the debugger through `dsdump --examine` the program will read the next event and start the debugger again. If you were running in client-server mode, the server will remain running, and you can reconnect to it and resume where you left off by entering '`ds db`' again. To kill the server, use the command '`ds kill`'.

A GUI front end to the DSPACK debugger is also available via the command '`dsmgr`'. You must have tcl/tk available for this to work. You should also have a DSPACK server running before starting `dsmgr`.

6.3 evdump

The `evdump` program can be used to dump out the header information in an EVPACK file. No information about the event contents will be displayed.

Here is an example of the output from the program:

```
At 0 Type: DSPACK definitions  Key: Run number: 1, Event Number: 1
  Reclen: 31395  Comp flag: 5  Uncompr size: 172032  Checksum: 1686699045
  Dictionary offset: 0
At 31395 Type: DSPACK data  Key: Run number: 1, Event Number: 1
  Reclen: 2978550  Comp flag: 5  Uncompr size: 8888320  Checksum: 3024682299
```

```
  Dictionary offset: 0
At 3009945 Type: DSPACK data  Key: Run number: 1, Event Number: 2
  Reclen: 2722989  Comp flag: 5  Uncompr size: 7700480  Checksum: 2251321869
  Dictionary offset: 0
```

The `evdump` program takes two arguments. If `--dump-index` is specified, the contents of any EVPACK index records will be printed. If `--dump-record` is specified, then a hex dump will be made of each record read (after it has been uncompressed).

6.4 evaddindex

The `evaddindex` program will read through an EVPACK file accumulating event key information. It will then discard any existing index record at the end of the file and write a new one. Any incomplete record at the end of the file will also be discarded. Thus, this program may be used to recover an EVPACK file that was not closed normally.

7 Unknown Objects

An "unknown" object is one for which there is no compiled-in type information; i.e., the linkage file wasn't linked in. This section describes the behavior of DØOM for such objects.

7.1 Output

Generally, when reference to an unknown object is encountered while writing, the object is ignored and a null reference is written instead. A warning is emitted when this occurs.

However, if the objects being written were read in from another file, special considerations apply. See Section 7.3 below for details.

7.2 Input

When DØOM finds a reference to an unknown object, it can dynamically build dictionary information for that class based on the type information in the data file. (Presently, this works only with the DSPACK back end, and only for data written by at least version `v00-15-00` of the library. For other cases, any references to unknown objects on input are simply set to null.) DØOM can then create objects based on this type information. However, any such objects will be instances of the type `d0_Unknown_Object`, and *not* instances of the C++ class to which the reference was referring. Therefore, dealing with instantiations of unknown objects requires special care, and this feature is disabled by default.

The behavior of DØOM when a reference to an unknown object is dereferenced is controlled by the `d0om_Options::unknown_action()`. This option has three possible values:

- `RETURN_NULL` — Return a null pointer. This is the default. If this happens, a warning will be generated by doing a `ZMthrow` of `Exc_Missing_Type`.
- `THROW_EXCEPTION` — Throw an `Exc_Missing_Type_Fatal` exception.
- `MAKE_UNKNOWN` — Create an object of type `d0_Unknown_Object`. This violates C++ typing rules, so you should enable this option only if you are prepared to deal with that.

The time at which this option is examined depends on the type of the reference. Bare C++ pointers (and `auto_ptr`'s) must be initialized with a C++ pointer; thus, this dereferencing occurs when the pointers are constructed. With the default setting of `RETURN_NULL`, a bare pointer pointing at an unknown object is initialized to null, and the unknown object is lost.

A `d0_Ref`, however, can be initialized to point at an unknown object. In this case, the option is not consulted until the reference is actually dereferenced. A `d0_Ref` pointing at an unknown object will have the following properties:

- `is_null()` is false.
- `is_unknown()` is true.
- `d0om_type()` returns the type of the unknown object.

If an attempt is made to reference the reference with `RETURN_NULL` in effect, a null C++ pointer will be returned.

If `MAKE_UNKNOWN` is enabled, any references which are read may actually be pointing at an instance of `d0_Unknown_Object`. Generally, the only safe way of accessing information in an object instance will then be through the dictionary information. Before this can be done, however, the pointer must be converted to a `d0_Object` pointer. This cannot be done safely using standard C++ pointer conversions; it should be done using the `conv_to_objptr()` method of `d0om_Type_Object`.

Most analysis code should not set `MAKE_UNKNOWN`. It is intended to be used by applications such as data structure browsers.

7.3 Copying

An additional feature is available for the case where a reference to an unknown object is encountered while writing and the unknown object was read from a file which contains type information. In that case, the object will automatically be instantiated for the write. After the write is complete, the object will be automatically deleted again.

The upshot is that when copying a structure from an input file to an output file, objects with no compiled-in linkage information can be copied too, provided that they are reached from known objects by `d0_Ref`'s and not bare C++ pointers. This option can be disabled with `d0om_Options::make_unknowns_on_write()`.

Note, however, the following limitation of the current implementation. If you copy events from several different input files which use different versions of an unknown

type, then the version used for *all* objects of that type in the output file will be that used in the *first* input file to use that type.

8 Schema Evolution

This section describes how class definitions may be changed so that old data files are still readable. Note that support for schema evolution depends on the implementation of the I/O backend. This section presently describes the status of the DSPACK backend, but the others should be similar.

In general, the commonest changes — including adding a new field — are handled automatically. Changing names or types, however, may require special attention.

Note that the schema evolution system described here only deals with changes in the *form* of the data. Changes in the *meaning* of the stored data cannot be handled without knowledge of the semantics of the class in question. If you must make such a change, you'll need to add a version number to your class, and handle the translation yourself.

8.1 Class Names

When reading a data file, DØOM matches classes which were saved to classes in the running program by matching names. This implies that if you change the name of a class, then you will not be able to read instances of that class in old data files. Note that "name" here refers to the fully-qualified class name — moving a class into a different namespace counts as changing its name! Therefore, you should think carefully about the name of a class and its namespace before writing data using it.

If, however, the name of a class must change, there is a feature which can help. In a header file processed through `d0cint`, include a directive like

`#pragma classalias` *newclass oldclass*`;`

This tells DØOM that the class presently known as *newclass* was previously known as *oldclass*. If DØOM encounters the class *oldclass* while reading a data file, it will map it to *newclass*. You may provide multiple `classalias` directives for a given *newclass*. You should not, however, attempt to create a data file containing both *oldclass* and *newclass*. Also, a `#pragma classalias` directive should probably be put inside a `#ifdef __D0CINT__` construction, in order to hide it from other language processors. (Or use `D0OM_CLASSALIAS`; see section 9.2.) (Note that, at present, only the DSPACK backend pays attention to this information.)

8.2 Class Members

DØOM translates between saved classes and classes in the program by matching the names of members. Therefore, members maybe freely added, deleted, or reordered. Note that, except as described below, renaming a class member is logically the same as deleting a member and then adding a new one.

If a member is deleted, the information contained in the data file for that member is silently ignored. If a member is added, it is initialized to zero when reading a data file which doesn't contain that member. There is no straightforward way to distinguish between a missing member and a member which just happened to be zero. If such a distinction is important for your class, we suggest that you add a version number member to the class.

Except as noted below, it is not allowed to change the type of a member. If that should happen, an exception will be thrown. This restriction may be relaxed in the future.

A numeric type may be changed into another numeric type. Note that a warning may be generated in cases where this loses precision. This is implemented as a set of predefined conversions (see Section 8.3).

If a member is an array, the size and dimensionality of the array may be changed freely, as long as the base type does not change. A non-array member may also be freely changed to an array, or vice-versa.

In addition, a collection type may be freely changed to any other collection type with the same element type. I.e., you can change `list<int>` to `vector<int>`, but not to `list<float>`.

Note that if a member is deleted from a class, that member name should not be reused in the future. (When reading an old data file, DØOM has no way of knowing that it's supposed to be a new member.) Therefore, when you delete members from a class, it is a good idea to preserve them in the class documentation. This is also helpful in understanding old data files.

If you must rename a persistent class member but need to be able to continue to read that member in old data files, you can use the `memberalias` directive, which works in a manner similar to `classalias`. It has the form:

> `#pragma memberalias` *newname oldname*`;`

Here, *newname* is looked up in the context in which the directive occurs. If *oldname* has a class or namespace qualifier, that is ignored. For example,

```
struct A { int a; };
#ifdef __DOCINT__
#pragma memberalias A::a b;
#endif
```

says that if the member 'A::a' isn't found in the data, DØOM should try looking for 'A::b' instead. As for `classalias`, this directive should probably be placed inside an `#ifdef __DOCINT__` construction. (Or use `DOOM_MEMBERALIAS`; see section 9.2.)

Note that, at present, the `memberalias` mechanism works only with the DSPSACK backend.

8.3 Conversions

The automatic rules DØOM uses for schema evolution suffice for many simple situations. However, some sorts of changes — particularly those that involve changes in

the meaning of the stored data — can never be done completely automatically. To help with such cases, DØOM allows arbitrary *converters* to be registered to convert between types.

A converter is an instance of an object that derives from `d0om::Converter<To>` (or `d0om::Converter_Base`). It knows how to convert between two types, here called the "source" and "target" types. If DØOM is expecting to see the target type but instead sees the source type, it does not throw an exception (as it would ordinarily do); instead, it runs the converter.

The target type of the conversion must be a type that is known to DØOM. The source type, however, need not be. In such a case, DØOM will automatically define a new type instance to describe the source type that it finds. The converter should then use the DØOM dictionary information to pick fields out of the source instance by name. An example of that will be given below. Due to the feature, the source type is always specified by name only.

Note that this section describes the case where the source and target types have different names. Conversions may also be used to convert between different versions of the same type; see section 8.3.3.

We'll first discuss how converters get registered, then turn to how to write converters themselves.

At this point, only the DSPACK backend handles conversions.

8.3.1 Registering Converters

There are two ways in which a converter may be registered: declaratively, by putting a `#pragma convert` directive in a header read by `d0cint`, or programatically, by calling `d0om_Dictionary::register_converter`.

A converter may be registered by putting a `#pragma convert` directive in a header being read by `d0cint`. The syntax is:

> `#pragma convert` *target-class source-type converter-name*

Here, *target-class* is the name of the class that is the target of the conversion. This must be a class that is known to `d0cint`, and it is looked up in the scope that is current where the directive appears.

source-type is the name of the source type of the conversion. This name need not be known to `d0cint`, and it need not be a class type. The name given is simply passed through to the `d0cint` output, so the name must be complete — including any namespace prefixes — and must not be a typedef name (or use any as template arguments).

converter-name is the name of the converter class. This must be a class deriving from `d0om::Converter` (or `d0om::Converter_Base`), and the class must have a default constructor. Again, this name is simply passed through to the `d0cint` output file, so it should include any namespace prefixes. This class must be defined in the linkage (`_lnk.cpp`) file, but it may be undesirable to include the header defining the converter directly from the header defining the class being converted, as that would mean that all users of that class have a dependency on the converter. This can be

avoided by using the `#pragma linkageinclude` directive, which emits its argument only in the linkage file. (See Section 9.1.)

Here's an example:

```
namespace foo {
  struct New_Class
  {
    int x;
  };

#ifdef __D0CINT__
# pragma linkageinclude "foo/Conv.hpp"
// New_Class is looked up in the current scope.
// Old_Class is not looked up here, so it needs
// to be a complete name.
# pragma convert New_Class foo::Old_Class Conv;
#endif
} // namespace foo
```

As mentioned earlier, a converter may also be registered programatically, by calling `d0om_Dictionary::register_converter`. See the `d0om_Dictionary.hpp` header for details. Here, you must pass it the converter instance. Here's an example:

```
  static Conv conv;
  d0om_Dictionary* g = d0om_Dictionary::global ();
  g->register_converter ("foo::Old_Class",
                         g->get_class_type ("foo::New_Class"),
                         conv);
```

The conversion to use should be unambiguous. If you register two conversions with the same source and target types with converters that have different dynamic types, you'll get a warning (the last one registered takes precedence).

Conversions are also considered when resolving pointers. Consider the following: you have a pointer declared to point to type `A`. Type `B` derives from type `A`. The pointer actually points to an object of type `C`, which is not known to DØOM, but there is a conversion from `C` to `B`. DØOM will resolve this by looking for any conversions from type `C` to any types that derive from `A`. If it finds more than one such candidate, it will issue a warning.

8.3.2 Writing Converters

A converter should derive from the class `d0om::Converter`, defined in the header `d0om/d0om_Converter.hpp`. (In some special cases, may be necessary to derive from `d0om::Converter_Base` instead.) Here are the relevant parts of the definition of `d0om::Converter`:

```
template <class To, class From=char>
class Converter
  : public Converter_Base
{
public:
  // Constructor, destructor.
  Converter () {}
  virtual ~Converter () {}

  // Convert FROM (of type FROM_TYPE) to TO (of type TO_TYPE).
  // The instance at TO will have already been constructed.
  virtual
  void convert (const From* from, const d0om_Type* from_type,
                      To*   to,   const d0om_Type* to_type) const = 0;
};
```

The derived class must implement the `convert` method. When it runs, it should convert the object at `from` to the object at `to`, assuming that the instance at `to` has already been constructed. DØOM passes to the converter the type instances for both the source and target types; this helps in writing generic converters.

Usually, the target type for a converter will be fixed at compile time. In that case, the `To` template parameter should be set to that type, and the `to` pointer passed to `convert` will already be cast to the correct type.

For the source type, though, there are two possibilities. The source type may also be known to DØOM. In that case, the `From` template argument may be set to the corresponding C++ type and the values manipulated directly. The other possibility is if the source type is not known to DØOM. In this case, the type of the source pointer should be left at its default, and the information from the source instance should be retrieved using the DØOM dictionary information. Examples of these two styles are given below.

First, consider the case where both types are known to DØOM. One might have, for example, these declarations:

```
struct Old_Class
{
  // Calorimeter cell index, 0-63.
  int iphi;
};


struct New_Class
{
  // Angle of center of calorimeter cell, 0-2pi.
  float phi;
};
```

```
#ifdef __D0CINT__
# pragma linkageinclude "foo/Conv.hpp"
# pragma convert New_Class Old_Class Conv;
#endif
```

The converter could then be written like this:

```
#include "d0om/d0om_Converter.hpp"
class Conv
  : public d0om::Converter<New_Class, Old_Class>
{
  virtual
  void convert (const Old_Class* from, const d0om_Type* from_type,
                      New_Class*   to, const d0om_Type* to_type);
};


#include "d0om/d0om_Dictionary.hpp"
#include <cmath>
#include <cassert>


void Conv::convert (const Old_Class* from, const d0om_Type* from_type,
                          New_Class*   to, const d0om_Type* to_type)
{
  // Check that we have the source type we expect.
  static const d0om_Type* expected_from_type = 0;
  if (expected_from_type == 0) {
    d0om_Dictionary* g = d0om_Dictionary::global ();
    expected_from_type = g->get_class_type ("Old_Type");
  }
  assert (from_type == expected_from_type);

  to->phi = from->iphi / 32. * M_PI + (M_PI/64);
}
```

The second style of converter, which does not assume that the source type is known to DØOM, is appropriate if you do not want you program to have any dependencies on the old code. In that case, the declaration of `Old_Class` in the example above could be dropped, and the converter written like this:

```
#include "d0om/d0om_Converter.hpp"
class Conv
  : public d0om::Converter<New_Class>
```

```
{
  virtual
  void convert (const      char* from, const d0om_Type* from_type,
                       New_Class*   to, const d0om_Type* to_type);
};


#include "d0om/d0om_Type_Class.hpp"
#include <cassert>


void Conv::convert (const      char* from, const d0om_Type* from_type,
                           New_Class*   to, const d0om_Type* to_type)
{
  // Get the source type descriptor as a d0om_Type_Class.
  // Check that we have the source type we expect.
  const d0om_Type_Class* fcls =
    dynamic_cast<const d0om_Type_Class*> (from_type);
  assert (fcls != 0);

  // Get the value of member 'iphi' in the instance
  // of 'fcls' at 'from'.
  int iphi = fcls->get_int (from, "iphi");

  // Do the conversion.
  to->phi = iphi / 32. * M_PI + (M_PI/64);
}
```

See the DØOM sources for further information about using the dictionary information.

8.3.3 Version Conversions

Converters can also be used between different versions of the same class. For this to work, you must have declared the version of the class to DØOM with a `#pragma version` directive (see section 2.9). You declare a converter from version n of class C to the current version n the same manner as described above, except that for the source name you use the special form "C-vn". This converter is used when reading any instances of version n and earlier, unless there is another converter registered with a smaller version.

For example, suppose we have the declarations

```
struct A {};
#pragma version A 6;
#pragma convert A A-v2 Conv1;
#pragma convert A A-v4 Conv2;
```

In this case, the converter `Conv1` will be used for versions 1 and 2, `Conv2` will be used for versions 3 and 4, and the default conversion rules will be used for any later versions. (Note that if the version read is *larger* than the current version, the default conversion rules are always used.)

8.3.4 Implicit Conversions

DØOM provides some implicit conversions, in addition to those that have been explicitly registered.

If there is a conversion from type `A` to type `B`, then there is an implicit conversion from any collection of `A` to any collection of `B`.

8.3.5 Predefined Conversions

DØOM registers some conversions by default.

There are six predefined conversions, between all of `int`, `float`, and `double`. However, the three of these that narrow the type give a warning when they run.

8.4 nowrite

There is an additional mechanism available that may be helpful when changing class members. A member may be tagged as "no-write," using `#pragma nowrite`, which works in a manner similar to `#pragma transient`. For example:

```
struct foo
{
  int a;
};

#ifdef __DOCINT__
# pragma nowrite foo::a;  // foo::a should not be written.
#endif
```

Class members that are tagged as `nowrite` will be read in from input files, if they exist, but will not be written to output files. This can be used to assist in schema evolution, as sketched in the following example.

Suppose you have a class foo:

```
struct foo
  : public dO_Object
{
  A a;
  DO_OBJECT_SETUP (foo);
};
```

You want to change 'A a' to 'B b', while retaining the ability to read old data files. One solution would be to maintain both members and add a conversion, as outlined here:

```
struct foo
  : public d0_Object
{
  A a;
  B b;

  void activate ();
  D0_OBJECT_SETUP (foo);
};

void foo::activate ()
{
  if (a.has_info() && !b.has_info()) {
    convert_a_to_b (a, b);
    a.clear();
  }
}
```

This will work, but it has the disadvantage that the declaration "`A a`" will take up some space in the output file. This overhead may be avoided by declaring the member `a` as `nowrite`:

```
#pragma nowrite foo::a;
```

8.5 Other Points

DØOM pays no attention to methods. Therefore, they may be freely changed, without affecting the ability to access stored data. The same applies to static data members and data members declared as transient.

DØOM expands all typedef names before writing to a data file. Therefore, typedef aliases for types may be freely changed.

9 `d0cint` Pragma Reference

9.1 Pragma Listing

The section lists some of the pragma directives accepted by `d0cint`. (`cint` proper recognizes some additional directives, which are probably not generally useful and are thus not documented here.)

These directives should all be on a single source line. They may optionally be ended with a semicolon, but that is not required. If they are used in source files

which are to be read by a C++ compiler, they should probably be put inside an `#ifdef __D0CINT__` construction, in order to hide them from the compiler. (Or use the macro variants listed in section 9.2.)

- `#pragma classalias` *newclass oldclass*
  Declare that *newclass* was previously known as *oldclass*. See Section 8.

  Example:

```
#ifdef __D0CINT__
# pragma classalias ns::foo foo;
  // Class 'ns::foo' used to be known as just 'foo'.
#endif
```

- `#pragma extendclass` *class*
  This pragma allows one to add to the definition of an already existing class. It should be followed on the next line by an open brace, then the body of the material to be added, then a closing brace. Note that the member protection is reset to the default; thus, when extending a `class`, all members will be considered private unless you include a `public:` keyword.

  This construction can be useful for injecting typedefs required by DØOM into classes defined in existing header files.

  Example:

```
class foo {};

#ifdef __D0CINT__
# pragma extendclass foo
{
public:
  // Inject this d0om typedef into class foo.
  typedef foo_adapter d0om_collection_adapter;
}
#endif
```

- `#pragma include_next "`*header*`"`
  This pragma searches the include path supplied to `d0cint` for *header*. Unlike `#include`, however, it does not stop at the first match it finds, but continues searching until it finds a *second* match. (Any paths containing '`/d0cintinc/`' are also skipped.) That header is then included. This is useful for writing `cint` wrapper headers around existing headers.

  Note also that this directive accepts only the `""` form of inclusion, not `<>`, and that it searches *only* the directories explicitly specified from the command line with `-I` switches.

  Example:

```
#pragma include_next "Package/foo.hpp"
```

- `#pragma linkage` *classes*
  Tell `d0cint` to generate linkage information for *classes*. This only has an effect if it is in the outermost header file. See Section 5.9.

  Example:

```
#ifdef __D0CINT__
# pragma linkage foo;    // Generate linkage info
                         // for class foo.
#endif
```

- `#pragma linkageinclude` *text*
  Have `d0cint` emit `#include` *text* in the linkage file. See Sections 2.2.5 and 2.15 for usage examples.

  Example:

```
#ifdef __D0CINT__
# pragma linkageinclude "foo.hpp"
  // Include foo.hpp from the linkage source.
#endif
```

- `#pragma memberalias` *newname oldname*
  Declare that the class member *newname* was previously known as *oldname*. See Section 8.2.

  In this construction, *newname* is looked up in the context in which the directive appears. Any class or namespace qualifiers present in *oldname* are ignored. The class must have a persistent member named *newname*, and must not have one named *oldname*.

  Example:

```
struct A { int a; };

#ifdef __D0CINT__
# pragma memberalias A::a b;
  // Member 'A::a' used to be known as 'A::b'.
#endif
```

  This tells DØOM that if it can't find the member `A::a` in the data, it should also try looking for `A::b`.

- `#pragma nolinkage` *classes*
  Tell `d0cint` to not generate linkage information for *classes*. This only has an effect if it is in the outermost header file. See Section 5.9.

  Example:

```
#ifdef __D0CINT__
# pragma nolinkage foo;  // Don't generate linkage info
                         // for class foo.
#endif
```

- `#pragma nowrite` *members*
  Tell `d0cint` that *members* are should be read but not written. See Section 8.4.

  Example:

```
struct foo
{
  int a;
};

#ifdef __D0CINT__
# pragma nowrite foo::a;  // foo::a should not be written.
#endif
```

- `#pragma pack (`*packspec*`)` *members*
  Tell DØOM to attempt to pack *members* as described by *packspec*. See Section 2.18.

  Example:

```
struct foo
{
  int a;
  int b;
};

#ifdef __D0CINT__
// Pack a,b into a single int.
# pragma pack (nbits=16) foo::a, foo::b;
#endif
```

- `#pragma transient` *members*
  Tell `d0cint` that *members* are transient. See Section 2.7.

  Example:

```
struct foo
{
  int a;
};

#ifdef __D0CINT__
# pragma transient foo::a;  // foo::a is transient
#endif
```

- `#pragma version` *class version*
  Tell `d0cint` that class *class* has version *version*. The name *class* is looked up in the context in which the directive appears. The version number *version* is an expression that evaluates to a constant integer. It is an error to attempt to alter the version number of a class. (Multiple `#pragma version` directives for a class are permitted as long as the version numbers are the same.) See Section 2.9.

  Example:

```
struct foo
{
  int a;
};

#ifdef __D0CINT__
# pragma version foo 5;
# pragma version foo 3+2;  // Ok -- same version number.
#endif
```

9.2 Pragma Macros

The header `d0om/d0om_pragma_macros.hpp` contains alternate, macro versions of most of the `d0cint` pragma directives. The advantage is that the macros can be defined to expand to something harmless if `d0cint` is not running; thus, you can avoid cluttering your source with `#ifdef __D0CINT__` directives. For example, instead of writing

```
...
#ifdef __D0CINT__
# pragma transient A::b;
#endif
```

you can write

```
#include "d0om/d0om_pragma_macros.hpp"
...
D0OM_TRANSIENT (A::b);
```

Here is the list of defined macros:

- `DOOM_CLASSALIAS(newcls,oldcls)`
- `DOOM_CONVERT(to,from,conv)`
- `DOOM_LINKAGE(name)`
- `DOOM_LINKAGEINCLUDE(path)`
- `DOOM_MEMBERALIAS(newmem,oldmem)`
- `DOOM_NOLINKAGE(name)`
- `DOOM_NOWRITE(name)`
- `DOOM_PACK(packspec,field)`
- `DOOM_TRANSIENT(name)`
- `DOOM_VERSION(cls, vers)`

A Some `d0_Ref<T>` Details

This section contains some of the details of how class `d0_Ref<T>` is implemented and how it interacts with the class `d0_Object`. The class diagram for `d0_Ref<T>` is shown in Fig. 8. Observe that, in addition to being associated with the class `d0_Object`, `d0_Ref<T>`'s are associated with another class `d0om_Indptr`, or "indirect pointers." There can be at most one `d0om_Indptr` associated with any instance of `d0_Object`. An object's reference count, if any, is part of its associated `d0om_Indptr`. There are several way in which the classes shown in Fig. 8 can be instantiated.

1. Null references are not associated with any instance of `d0_Object` or any indirect pointer.
2. Static and automatic `d0_Object`'s have no associated indirect pointer or reference count. Such non-reference-counted objects may still be pointed to by a `d0_Ref<T>`.
3. Instances of `d0_Object` that are created on the heap using the `new` operator are associated with one indirect pointer of type `d0om_Transient_Indptr` and one or more `d0_Ref<T>`'s.
4. Except as provided for in the following case, objects read from DSPACK are associated with one indirect pointer of type `d0om_DS::Indptr` and one or more `d0_Ref<T>`'s. Instantiation of the corresponding `d0_Object` is deferred until one of the associated `d0_Ref<T>`'s is dereferenced.

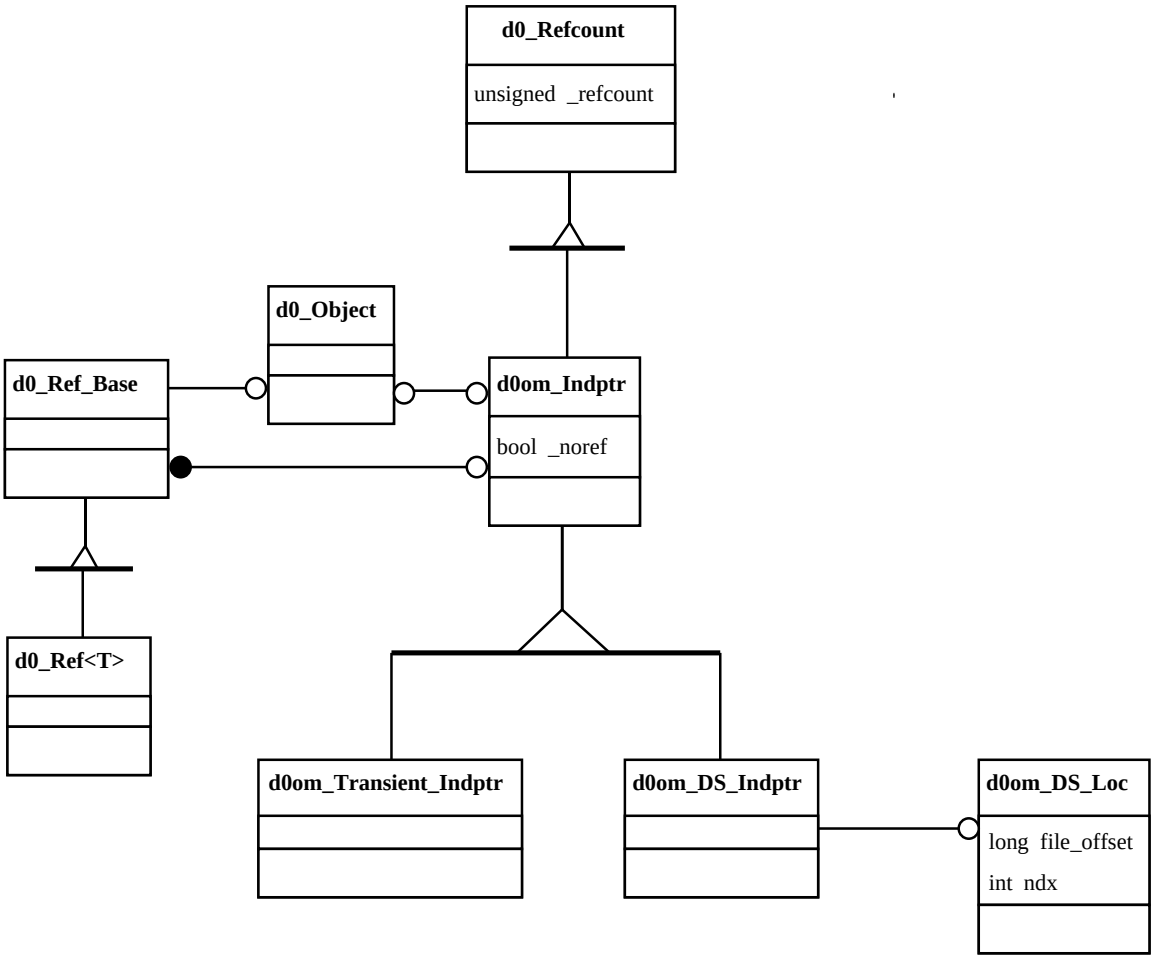

Figure 8: Class diagram for `d0_Ref<T>`.

5. Objects that have been read from DSPACK using bare C++ pointers have an associated indirect pointer of type `d0om_DS::Indptr`, but are never allowed to be pointed to by a `d0_Ref<T>`. DØOM enforces this by setting the flag `_noref` in `d0om_Indptr` to be true. In this case, the instantiation of the corresponding `d0_Object` can not be deferred, as it would be for `d0_Ref<T>`.

Cases three and four represent the typical non-trivial uses of `d0_Ref<T>`. In either case, the number of `d0_Ref<T>`'s that are pointing to an indirect pointer is maintained in the reference count that is part of the indirect pointer. Should the reference count ever reach zero, the indirect pointer and its associated `d0_Object` are deleted.

If I/O methods other than DSPACK are added to DØOM, it will be necessary to create additional subclasses of `d0om_Indptr`, as well as of `d0Stream`.

B DØOM Dictionary Classes

This section summarizes DØOM's dictionary classes. A diagram of the dictionary classes is shown in Fig. 9. Each C++ data type or subtype is described by a class

derived from abstract class `d0om_Type`. Each DØOM type is identified by a character string name and by subtype. Each DØOM subtype falls into one of the following four categories, each of which corresponds to one of the subclasses of `d0om_Type`.

1. Atomic types.
2. Collections.
3. References.
4. Classes.

In addition to these classes which describe types, there is an additional class `d0om_Dictionary` which manages the type instances. A `d0om_Dictionary` instance owns a collection of type instances, and has methods for creating them and looking them up.

There is one distinguished `d0om_Dictionary` instance, called the "global" dictionary. It can be found using the static method `d0om_Dictionary::global()`. Other dictionaries are "local." When a local dictionary is searched for a type, if the type isn't found, the global dictionary is automatically searched as well.

The type objects for classes and objects are created in the (static) routine `d0om_type_setup()` in each linkage source file. This gets called during initialization (from `d0om_init`).

B.1 Atomic Types

Atomic types are one of the types listed in Section 2.1, and are described by class `d0om_Type_Atomic`. Class `d0om_Type_Atomic` adds a single enumerated datum to `d0om_Type` which identifies the atomic type.

B.2 Collections

Collection types are one of the containers listed in Section 2.2, and are described by class `d0om_Type_Collection`. Class `d0om_Type_Collection` has as its data an enumerated collection type, and a link to the contained type, which can be any DØOM type.

B.3 References

Class `d0om_Type_Reference` is used to describe bare C++ pointers, bare C++ references, and `d0_Ref<T>`'s. Class `d0om_Type_Reference` contains an enumerated reference type and a link to the pointed to type, which must derive from `d0_Object`, and which is described by class `d0om_Type_Object`.

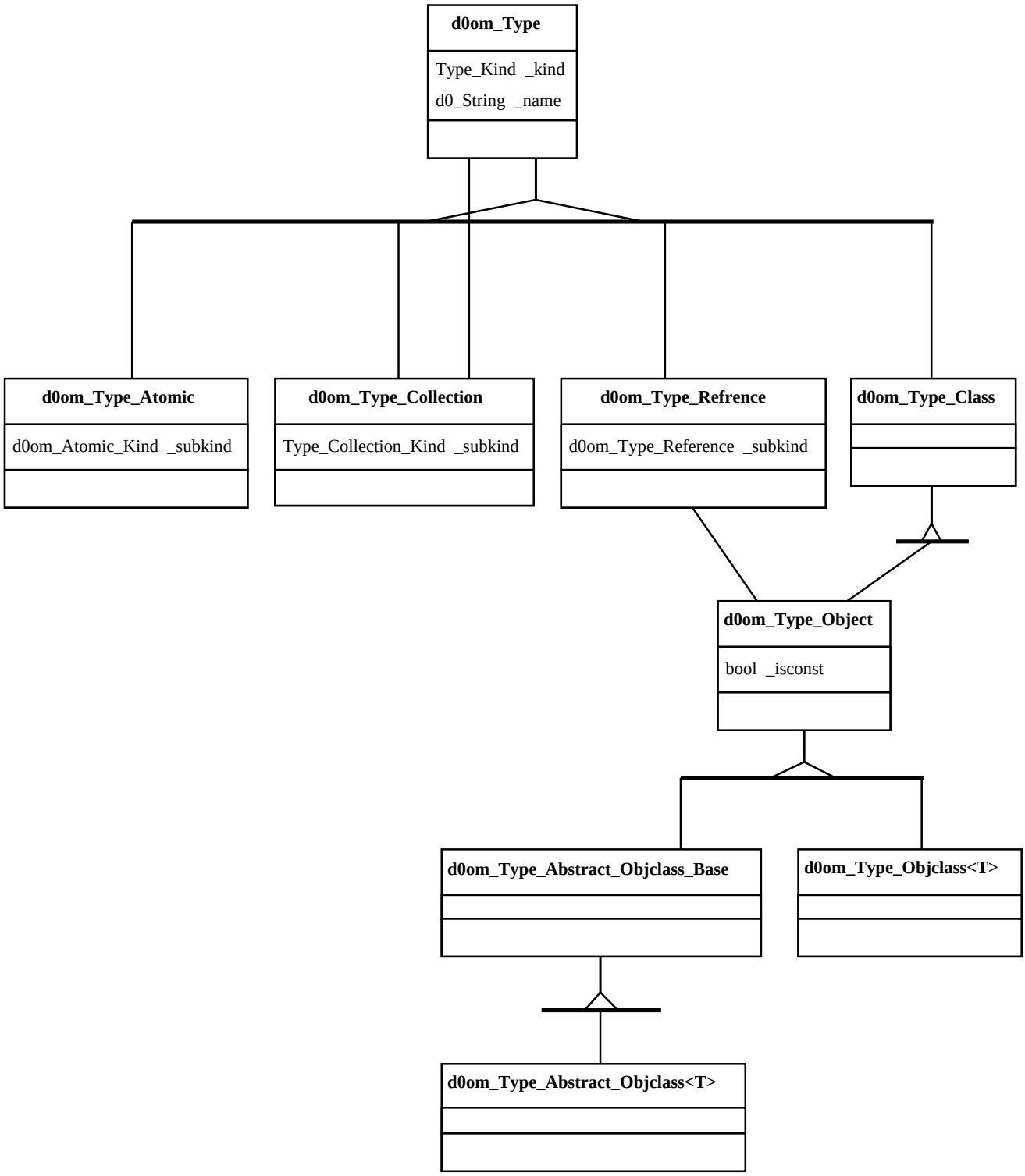

Figure 9: Class diagram for dictionary classes.

B.4 Classes

DØOM classes are described by class `d0om_Type_Class`, whether or not they are derived from `d0_Object`. Information about the base classes and data fields that make up the class are stored in class `d0om_Type_Class` itself (details not shown). Classes that don't derive from `d0_Object` are instantiated as a concrete instances of class `d0om_Type_Class`. Classes that derive from `d0_Object` are instantiated as subclasses of class `d0om_Type_Object`. Concrete and abstract classes are described respectively by instances of the template classes `d0om_Type_Objclass<T>` and `d0om_Type_Abstract_Objclass<T>`, where the template argument `T` is the type of the class being described.

B.5 Accessing the Dictionary classes

The complete dictionary interface is too complicated to completely document here. And end users should not normally need to access the dictionary. However, this section contains some hints about how to gain access to dictionary information.

The following methods of `d0_Object` allow the programmer to determine the type of any instance of a class that is derived from `d0_Object`.

```
virtual const d0om_Type_Object* d0om_type() const; // Dynamic type
static d0om_Type_Object* d0om_type_static();       // Static type
```

The class `d0om_Object_Walker` uses the dictionary information to visit each data member of a class instance and perform user-specified processing on them. See `d0om_ds/src/utils/dsdump.cpp` for an example of its use.

C DSPACK Specific I/O Interface

This section documents the I/O interface presented by the DSPACK interface classes. This interface is not intended to be used by ordinary users. It is included here for completeness.

Writing is handled by class `d0om_DS::Outunit` (which is defined by the header file `d0om_ds/Outunit.hpp`). Reading is handled by class `d0om_DS::Inunit` (header file `d0om_ds/Inunit.hpp`) The following program fragment shows the steps involved in writing an event.

```
#include "d0_util/d0_String.hpp"
#include "d0om/d0_Object.hpp"
#include "d0om/d0_Ref.hpp"
#include "d0om_ds/Outunit.hpp"
#include "Event.hpp"

int main()
{
```

```
  d0_Ref<Event> rEvent;

// System initialization.  The argument name is passed to DSPACK's
// initialization routine, dsinit.

  d0om_init("test");

// Open output file

  d0om_DS::Outunit* oun = d0om_DS::Outunit::open("test.ds");

// Write event.  Every class referenced in Event is also written out.

  oun->write_obj (rEvent);

// Close output file.

  oun->close();   // Close output file
}
```

The following is an example of a reading program.

```
#include "d0_util/d0_String.hpp"
#include "d0om/d0_Object.hpp"
#include "d0om/d0_Ref.hpp"
#include "d0om_ds/Inunit.hpp"
#include "Event.hpp"

int main()
{
  d0_Ref<Event> rEvent;

// System initialization.

  d0om_init("test");

// Open input file.

  d0om_DS::Inunit* iun = d0om_DS::Inunit::open("test.ds");

// Event reading loop.  Read a new DSPACK record.

  while(iun->read_dsrec()) {
```

```
// Return the first object of class Event in the current record.

    rEvent =
      D0_REFCAST((Event)) (iun->getref(1, Event::d0om_type_static()));
  }

// Close input file.

  iun->close();
}
```

C.1 DØOM to DSPACK Interface Classes

This section contains a more detailed description of the classes which provide the mapping between the DØOM data model and its I/O interface; and the data structures, files and I/O mechanisms of the DSPACK package. The DØOM stream I/O interface is abstract and therefore independent of any particular I/O mechanism. However, each specific stream implementation requires a set of interface classes through which the actual translation of data between C++ objects and the data structures of the I/O mechanism occur and through which the mapping between an I/O stream and the physical file I/O mechanisms is accomplished.

The interface between DØOM and DSPACK consists of the following elements

1. Specific sub-class of d0Stream for DSPACK I/O
2. Mapping classes between DØOM Persistent classes and DSPACK datasets, including a class which represents a DSPACK directory
3. Pointer mapping classes
4. DSPACK File Identifier representation classes
5. DSPACK class through which all calls to DSPACK are made
6. Exception handling

C.2 d0StreamDSPACK

This sub-class of `d0Stream` currently reads and writes events from a `d0Stream` which has been mapped to a DSPACK data file. An event in a DSPACK data file consists of a number of interdependent DSPACK datasets, treated as a single logical record.

However, in addition to this basic read/write file stream I/O, an additional stream read/extract capability is provided - to return a pointer to a named class based on a key (which maps one-to-one onto a DØOM class name). This type of read/extract is only performed when the key, a character string, consists of a minus sign (-) followed by the name of a class.

The implementation also allows for an "event" to be written out consisting of any tree of DØOM persistent objects starting from a specific `d0_Object`. Checks for circular pointer references are implemented.

Creation of a `d0StreamDSPACK` instance returns an open stream behind which all of the needed objects for mapping between stream I/O and DSPACK I/O and between C++ classes and DSPACK datasets have been created and populated. This involves creation of an appropriate `d0om_DS::Unit` instance (either for input or output) and creation of a matching "Directory" object for the stream. The "Directory" object holds various DSPACK indices and also manages the collection of objects which serve to provided data mapping functions - see Saveable classes described below.

A `d0StreamDSPACK` object when instantiated :-

1. opens a DSPACK data file, for either read or write, using a `d0om_DS::Unit` class object as the representation of the DSPACK input or output file and its file identifier.

2. creates a directory object to store and manage the map between DSPACK dataset objects and C++ classes. If data is being read then the DSPACK header file, which contains 'self-defining' complete definitions of all objects that may be found in the file, is first read and used to construct the directory.

3. creates instances of Saveable classes - one for each type of persistent object to be mapped between DSPACK and DØOM

C.3 I/O stream to file ID mapping classes

The class `d0om_DS::Unit` allocates and manages DSPACK file IDs (or unit numbers as they are sometimes called). There are specialized sub-classes for input units and output units.

```
d0om_DS::Inunit (subclass of d0om_DS::Unit)
d0om_DS::Outunit (subclass of d0om_DS::Unit)
```

C.4 Mapping classes between DØOM persistent classes and DSPACK datasets

C.4.1 d0om_DS::Dir

This class, together with its partner class `d0om_DS::Dirrep`, forms an object registry for DSPACK datasets. It tracks the object index, DSPACK handle, section location in DSPACK for each object. It creates Saveable objects for each type of object being managed. It keeps a map of complete descriptive information for each field of each object.

A class `d0om_DS::Dirchange` provides a mechanism for changing the directory for a `d0StreamDSPACK` stream.

A class `d0om_DS::Rawdir` is used to manage the creation and allocation of DSPACK directories.

C.4.2 Saveable Base Class

`d0om_DS::Saveable` is the base class for classes which play the role of 'Agents' in the conversion of data between DØOM C++ class instances and elements in DSPACK datasets. A `d0om_DS_Saveable` instances is associated with a particular DSPACK directory. For each directory and for each type of DØOM persistent class present a `d0om_DS::Saveable` sub-class of matching type is instantiated and used to carry out the mapping and data conversion operations.

The base class handles the common functions needed for all DØOM persistent classes, namely:

- the generic mapping from the C++ class name to a valid DSPACK dataset name. This may involve mangling the class name to conform to DSPACK names.
- access to DSPACK dataset handles
- the generic movement of N data items between the C++ memory location and the corresponding DSPACK memory buffer, including reservation of space in the DSPACK memory buffer. These methods, which save and restore data, are normally overridden by subclass methods.

Subclasses of `d0om_DS::Saveable` handle the specific data translation functions involved in mapping each type of DØOM persistent class to the elements of its DSPACK dataset representation.

C.4.3 d0om_DS::Class and d0om_DS::Classrep

Saveable subclasses for different types of DØOM persistent classes.

These two classes together hold the mapping between C++ class member fields and DSPACK fields for a particular class. They are logically one mapping class, but are sub-divided into two classes because of compiler/template limitations. Each C++ class member field is represented in a map table by

- field name.
- offset in the C++ structure.
- number of adjacent identical elements in the C++ structure.
- offset in the DSPACK structure (or -1 if doesn't exist in DSPACK).
- number of adjacent identical elements in the DSPACK structure.
- the type of Saveable for this member field.
- flag for if the field represents a base subobject.

`d0om_DS::Class` methods which save and restore data use the mapping table to determine how to allocate space and initialize memory. They then perform the actual translation and copy of the fields.

`d0om_DS::Object` is a subclass of `d0om_DS::Class`. All C++ classes which inherit from `d0_Object` are mapped and translated by the subclass `d0om_DS::Object` of `d0om_DS::Class` which, in addition to calling the base class methods to save and restore, also performs the activation and deactivation of an object instance as it is read in (restored) or written out (saved).

C.4.4 d0om_DS::Collection

This subclass handles the save and restore of the data structures representing a DØOM Container (aka Collection) class and also iterates over the elements of the Container, invoking the save and restore methods of the appropriate Saveable type for each of the contained instances. All Container collections are represented by two separate DSPACK datasets. One dataset contains a two element structure denoting the number of items in the collection and a pointer to the first collection element. The other dataset stores the actual collection elements contiguously. All DØOM Container types are handled in the same way. For ordered collections the order of the elements is significant.

C.4.5 d0om_DS::Atomic

This subclass, and its more specific derived classes, handle the mapping of DØOM atomic data types. The fixed length fundamental atomic data types - Integer, Short, Character, Boolean, Float and Double are all handled in a similar way by the `d0om_DS::Fundamental` classes. These handle the translation of the fundamental atomic types between a C++ field and the DSPACK dataset named for the specific data type - e.g. ".INT4" for Integers.

All supported fundamental atomic types except for Double require no further code to perform the mapping beyond that provided through the C++ language support. The one exception to this is Double, for which a supporting class `d0om_DS_double_hack` is needed.

C.4.6 d0om_DS::String and d0om_DS::Stringrep

The mapping between a C++ atomic field and a DSPACK dataset element is handled slightly differently for string atomic types. A string is represented in two separate datasets; one storing a two-element structure containing a pointer to the start of the string and its length, and the other storing the actual body of the string. The `d0om_DS::Stringrep` class merely provides the mapping to named DSPACK dataset "x.CHAR4" which is used for storage of the body of strings - packed into an integral number of 4 byte character string elements.

C.4.7 d0om_DS::Reference

The mapping between a C++ member field which stores either a C++ bare pointer or a `d0_Ref` pointer is handled by this class or one of its derived classes

- `d0om_DS::Reference_Pointer`: for handling C++ pointers or references.
- `d0om_DS::Reference_D0Ref`: for handling `d0_Ref` objects.
- `d0om_DS::Reference_DynRef`: for handling `DynRef` objects.
- `d0om_DS::Reference_Auto_Ptr`: for handling `auto_ptr` objects.

All types of references contained in C++ classes are implemented in their DSPACK representation as DSPACK pointers. The mapping between references and DSPACK pointers is quite tricky and has many checks to

- ensure that there are no circular pointer references
- handle polymorphism
- check for potential conflicts between C++ bare pointers and DØOM smart pointers

C.4.8 d0om_DS::Dummy

Although this class is derived from `d0om_DS::Saveable` it does not carry out mapping functions in the same way as the other sub-classes. Rather, it is used by those other classes when, in the course of mapping between a DSPACK dataset and C++ classes, it is discovered that a particular dataset contains a field which has no corresponding field in its C++ class. An instance of a `d0om_DS::Dummy` class is then created by the Saveable class handling the mapping. This instance is used merely as a temporary memory storage location so that the normal mapping operations of locating space, clearing/zeroing space, and performing a copy can proceed into a "dummy" data area. An instance of `d0om_DS::Dummy` is not associated with a particular DSPACK directory; it is created and destroyed as needed by the `d0om_DS::Saveable` instance.

C.5 Location mapping classes

C.5.1 d0om_DS::Loc

Objects are located in DSPACK using a triple

```
(file_offset,Saveable,index)
```

This triple is represented by the class `d0om_DS::Loc`.

C.5.2 d0om_DS::Indptr

This class is a DSPACK specific subclass of `d0om_Indptr` which provides the methods to locate an object in memory using `d0om_DS::Loc` objects.

C.5.3 d0om_DS::Bound_Pointer

???

D Cint License

License:

- License condition described in this README file overrides other descriptions if there is any difference.
- Copyright of Cint and associated tools are owned by Hewlett-Packard Japan Company and the author. Acknowledgement to the author by e-mail is recommended at installation.
- Source code, binary executable or library of Cint and associated tools can be used, modified and distributed free of charge for non-commercial purpose provided that the copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Registration is requested, at this moment, for commercial use. Send e-mail to the author. The registration is free. (`gotom@jpn.hp.com` or `MXJ02154@niftyserve.or.jp`)
- If a modification is made on any of the source or documentation, it has to be clearly documented and expressed.
- Hewlett-Packard Japan Company and the Author make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.
- Above condition will overrides other license described in source code and other documentation if there is conflict. It applies and will not be changed for this revision of CINT package.
- Support and consulting of CINT will be available from Hewlett-Packard Japan "MPN Consulting Group". CINT is distributed as free software, however, there have been requests for commercial uses and there are people who feel comfortable about having official support channel. Contact `Akira_Fujita@om.jpn.hp.com` or the author(`gotom@jpn.hp.com`). Having official support and consulting contract is recommended for serious commercial project.

- readline, glob and malloc directories contain files associated with GNU readline library which is copylefted by GNU project. Refer to General Public License(GPL). Cint and the GNU readline are completely separate software packages which can work independently.
- For copyright notice and licensing of AIX dlfcn, please read platform/aixdlfcn/README.

References

[1] R. Zybert, DSPACK User's Guide, version 1.12.

[2] *The Object Database Standard: ODMG-93, Release 1.2*, R.G.G. Cattell ed., Morgan Kaufmann, San Francisco, 1996.

[3] `http://root.cern.ch/`

[4] `http://www-d0.fnal.gov/software/cmgt/cmgt.html`

[5] *Design Patterns*, E. Gamma *et al.*, Addison-Wesley, Reading, Mass., 1995.